ABSTRACT

ZHANG, YONGPENG. Exploiting Data-Parallelism in GPUs. (Under the direction of Dr. Frank Mueller.)

Mainstream microprocessor design no longer delivers performance boosts by increasing the processor clock frequency due to power and thermal constraints. Nonetheless, advances in semiconductor fabrication still allow the transistor density to increase at the rate of Moore's law. This has resulted in the proliferation of many-core parallel architectures and accelerators, among which GPUs (graphics processing unit) quickly established themselves as suitable for applications that exploit fine-grained data-parallelism. GPU clusters are starting to make inroads into the HPC (high performance computing) domain as well, due to much better power per Flop (floating point operation) performance than general-purpose processors such as CPUs.

Even though it is easier to program GPUs than ever, efficiently taking advantage of GPU resources requires unique techniques that are not found elsewhere. The traditional function level task-parallelism can hardly provide enough optimization opportunities for such parallel architectures. Instead, it is crucial to extract data-parallelism and map it to the massive threading execution model advocated by GPUs.

This dissertation consists of multiple efforts to build programming models above existing models (CUDA) for single GPUs as well as GPU clusters. We start from manually implementing a flocking-based document clustering algorithm on GPU clusters. With this first-hand experience to write code directly above CUDA and MPI (message passing interface), we make several key observations: (1) Unified memory interface greatly enhances programmability, especially in GPU cluster environment, (2) explicit expression of data parallelism at language level facilitates the mapping of algorithms to massively parallel architectures and (3) auto-tuning is necessary to achieve competitive performance as the parallel architecture becomes more complex.

Based on these observations, we propose several programming models and compiler approaches to achieve portability and programmability while retaining as much performance as possible.

- We propose GStream, a general-purpose, scalable data streaming framework on GPUs. We project powerful, yet concise language abstractions onto GPUs to fully exploit their inherent massive data-parallelism.
- We take a domain specific language approach to provide an efficient implementation of 3D iterative stencil computations on GPUs with auto-tuning capabilities.
- We propose CuNesl, a compiler framework to translate and optimize a nested data-parallel language called NESL into parallel CUDA programs for SIMT architectures. By converting recursive calls into while loops, we ensure that the hierarchical execution model in GPUs can be exploited on the "flattened" code.

- Finally, we design HiDP, a hierarchical data-parallel language suitable for hierarchical features of microprocessor architectures. We then develop a source-to-source compiler that converts HiDP into CUDA C++ source code with tuning capability. It greatly improves coding productivity while still keeping up with the performance of hand-coded CUDA code.

The methods above cover a wide range of techniques for GPGPU computing and represent the current technology trend to exploit data parallelism in state-of-the-art GPUs.

Exploiting Data-Parallelism in GPUs

by
Yongpeng Zhang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2012

APPROVED BY:

Dr. Xiaosong Ma	Dr. Nagiza Samatova
Dr. Huiyang Zhou	Dr. Frank Mueller Chair of Advisory Committee

DEDICATION

To Pei, Samuel and Grace.

BIOGRAPHY

Yongpeng Zhang was born and raised in Hunan Province, China. In 1997, He went to Beijing to attend Beihang University, where he received his undergraduate degree in Electrical Engineering. He continued to pursue the Master's degree in Drexel University, Philadelphia.

He returned to China after graduation in 2003 and worked in a couple of start-up companies. Working as a software engineer, he found himself more and more interested in high-performance and parallel computing.

In 2008, he decided to go back to school and joined North Carolina State University to pursue Ph.D degree in the Computer Science department. He was supervised by Dr. Frank Mueller and focused on applications, compiler frameworks and run-time systems for GPUs. He also worked at the Oak Ridge National Lab and Nvidia as a summer intern. He will be joining Stone Ridge Technology in Bel Air, MD after graduation.

ACKNOWLEDGEMENTS

This dissertation is not made in one day, nor by one person. It is only possible with the support and encouragement of many people, to whom I take this opportunity to acknowledge.

First of all, I would like to thank Dr. Frank Mueller, for being my academic advisor, mentor and friend. Working with him has been an invaluable and pleasant experience to me. His wisdom, knowledge and professionalism inspired and motivated me. His keen efforts to identify problems and brainstorm ideas have influenced my way of conducting research and will guide me through my future career.

I also want to thank my friends in the NCSU system lab: Abhik Sarkar, Chris Zimmer, Xing Wu, Feng Ji, David Fiala, Fei Meng, Arash Rezaei and James Elliott (hint: there is an ordering pattern here). The countless afternoon conversations on politics, technologies, food, games, sports, travel experience in other countries, Mac versus PC, Android versus IOs, life in general and anything interesting happened on that day have turned EBII 3226 a cultural melting pot. Thank you for making my life in NCSU as enjoyable as it is. Best wishes to you all.

I am indebted to my committee members: Dr. Xiaosong Ma, Dr. Nagiza Samatova and Dr. Huiyang Zhou. They have provided invaluable insight and suggestions to my presentations and draft.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

A graphics processing unit (GPU) is a specialized circuit designed to rapidly accelerate the building of images in a frame buffer for real-time output on a display. The term "GPU" was first used by Nvidia in 1999 marketing the GeForce 256 as the "world's first 'GPU', or Graphics Processing Unit" and widely adopted ever since. Though GPUs can be integrated with CPUs in the same die, more powerful GPUs are generally found on discrete GPUs, as a separate video card connected to the motherboard.

Because of their unique and relatively narrow application area, GPUs differ from general-purpose microprocessors (CPUs) in their architecture design from the ground up. Instead of trying to make sequential and general programs running faster and faster, GPUs are made to accelerate a sequence of operations on vertexes independently in a pipeline, a process that has become so standard that it can be accelerated by dedicated hardware. Due to the inherent parallelism of vertex shading, GPUs adopted multi-core architectures long before CPUs resorted to such a design. While in the former case, this decision is driven by increasing demands for faster and more realistic graphics effects, it is dictated by power and asymptotic single-core frequency limits for the latter.

1.1 On the History of GPGPU Programming

Today's state-of-the-art GPUs consist of many small computation cores compared to few large cores in off-the-shelve CPUs, at the cost of devoting less die area for flow control and data caching in each core. They deliver much higher raw performance in terms of GFlops. This has attracted many developers who strive to combine high performance, lower cost and reduced power consumption as an inexpensive means for solving complex problems. The history of using GPU as an alternate parallel desktop computing platform for non-graphics processing (GPGPU) can be roughly divided into three phases:

- Graphic APIs: In the early 1990s, the increasing demand for 3D real-time graphics called for standard application programming interfaces (APIs), among which OpenGL and DirectX became

the most popular. Exploiting GPU resource, at that time, could only be done via those graphics APIs. Only a very limited number of algorithms could be efficiently mapped into graphics APIs. [80] and [66] are a few successful examples.

- Programmable Shading: Nvidia was first to produce a chip capable of programmable shading (GeForce 3, 2001), where a short user-defined program can be inserted into certain stages in the graphics pipeline. Though it added flexibility to users, many constraints still applied to the shading programs, such as the limited length of the program, the number of registers and the supported instructions. Some of the constraints were loosened later, *i.e.* floating-point calculation was first added in 2002 by ATI with looping capability. The scope of applicable algorithms has been broadened to many new areas ([95] [98] [56] and [75]).

- CUDA and OpenCL: Programmable shading requires programmers to craft algorithms in a graphics context. It often takes graphics experts to do so. The launch of CUDA (Compute Unified Device Architecture) by Nvidia in 2006 (GeForce 8800) truly made GPGPU accessible to programmers in general. The new programming model no longer requires graphics as prior knowledge. The ease of programming catalyzed tremendous amount of research in accelerating applications in various areas [51, 92, 105, 42, 6, 63, 106]. Later, the Khronos Group defined an open standard, OpenCL, which gained support by Intel, AMD, Nvidia and ARM.

Today, GPU clusters are making inroads into HPC (high performance computing) domain, which was previously dominated by general-purpose processors. As of June 2011, GPUs are the major GFlops contributor for three clusters out of the top 10 most powerful supercomputers. Oak Ridge National Lab is planning to use the newest generation of GPUs to build a 20 Petaflop supercomputer in 2012. Programming on GPUs has never been as ubiquitous as today.

1.2 State-of-the-Art GPUs

In this section, we provide an introduction to the architecture and programming model for the state-of-art GPUs. We focus on Nvidia's GPUs and CUDA.

Today's GPUs have evolved to a customizable, highly parallel computing platform. An overview of the Nvidia Fermi architecture is shown in Figure 1.1. It features 512 CUDA cores organized in 16 Streaming Multiprocessors (SMs) of 32 cores each. Each CUDA core has a fully pipelined integer arithmetic logic unit (ALU) and floating-point unit (FPU). The GigaThread global scheduler distributes thread blocks to SM thread schedulers. The SM schedules threads in groups of 32 parallel threads called warps. Each SM contains two warp schedules and two instruction dispatch units, allowing two warps to be issued and executed concurrently. Each warp is assigned to a group of sixteen cores and sixteen load/store units.

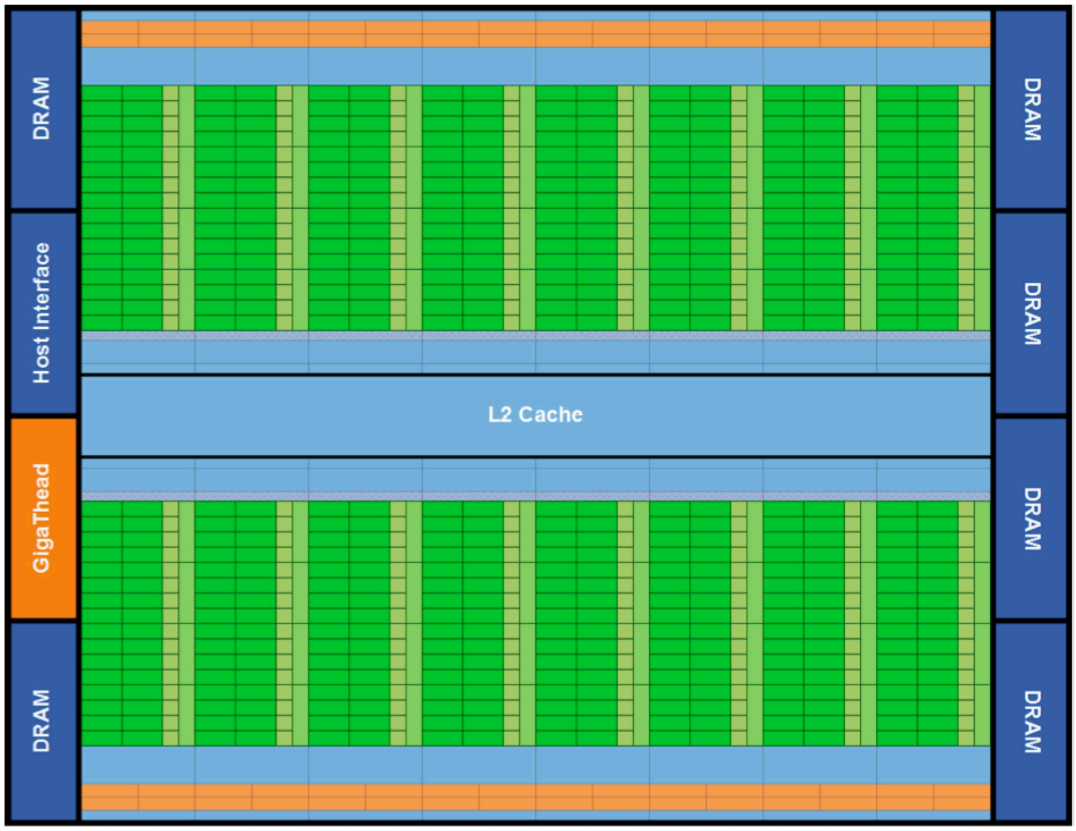


Figure 1.1: Nvidia Fermi Architecture (Source:NVIDIA)

A host interface connects the GPU to the CPU via the PCI-Express bus. The GPU has six 64-bit memory partitions for a 384-bit memory interface supporting up to a total of 6 GB of GDDR4 DRAM memory. The GPU memory hierarchy contains the following levels, from fastest to slowest (see Figure 1.2):

- Register File: In contrast to CPUs, GPUs maintain a much larger register file size of 32K-word per SM. Registers are dedicated to threads once they are assigned to them. This allows multi-thread context switches at clock cycle rate (in a single cycle).
- Configurable On-chip Shared Memory: In the Fermi architecture, each SM has 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache or as 16 KB of Shared memory with 48 KB of L1 cache. Shared memory can be treated as a scratch pad that gives programmers more control of data locality.
- L2 Cache: It is shared by all SMs in the chip. It also supports a set of memory read-modify-write operations that are atomic.

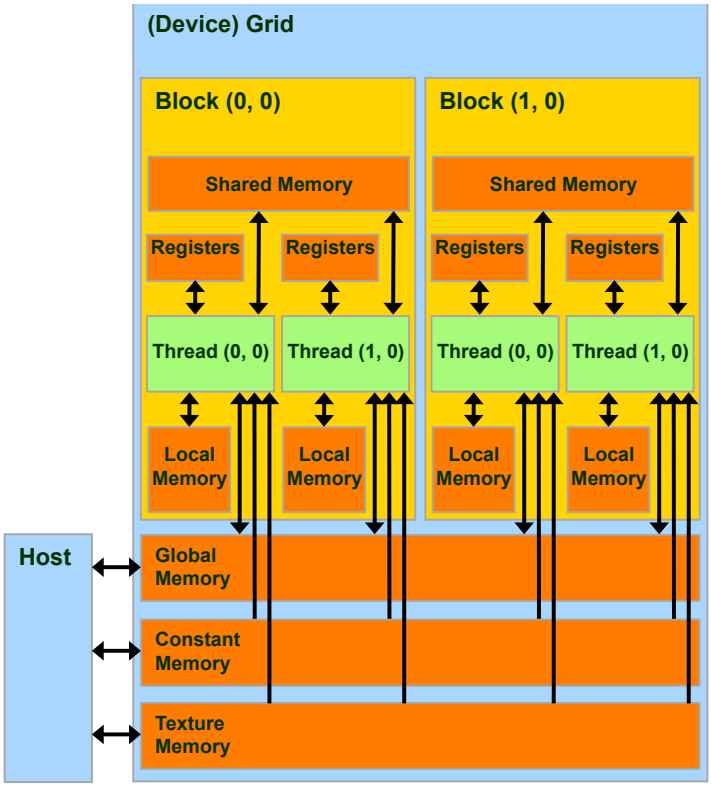


Figure 1.2: CUDA Memory Overview (Source:NVIDIA)

- Off-chip memory: This includes global memory, constant memory and texture memory. Constant memory and texture memory use a different cache than the L2 cache. It is sometimes beneficial to map data into constant or texture memory to avoid polluting the L2 cache.

All levels of memory are protected by ECC (error correcting code) in Tesla GPU cards. ECC can correct single bit errors and detect double bit errors.

On the software side, a CUDA program calls parallel kernels. A kernel executes in parallel across a set of parallel threads, which are mapped to different hierarchies, from top to bottom (see Figure 1.3):

- Grid: The GPU instantiates a kernel program on a grid of parallel thread blocks. One kernel maps to one grid of threads. It is at this top level that CUDA threads are created and destroyed. An implicit barrier exists between kernel calls.

- Block: A grid is organized as a set of 1D, 2D or 3D of blocks. A block has a unique ID. Threads inside one block can synchronize and share data through the usage of Shared memory.

- Warp: This is the GPU's instruction scheduling unit. A block can contain one or more warps. A warp of threads always executes the same instruction at the same time. Every code path of

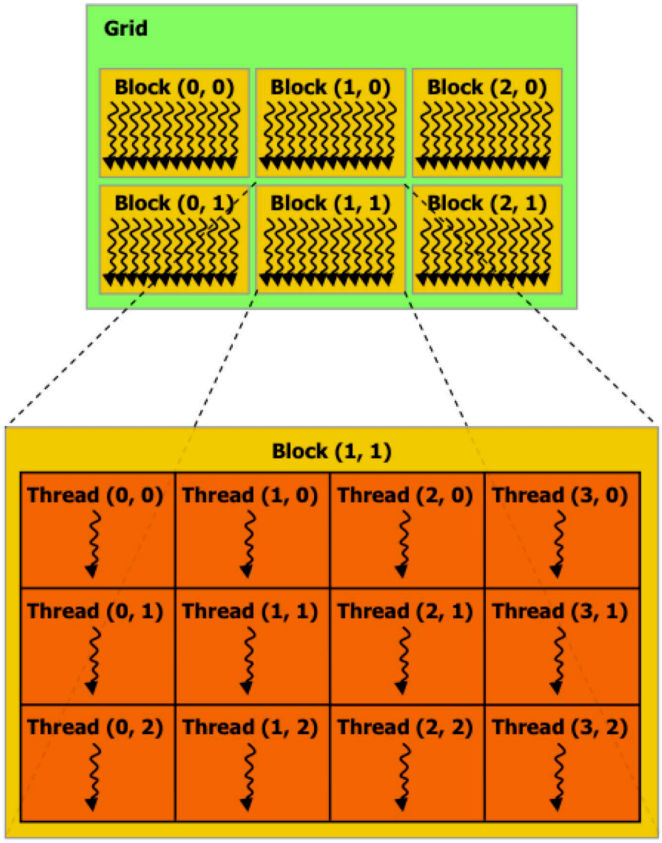


Figure 1.3: CUDA Thread Hierarchy (Source:NVIDIA)

branches needs to be executed if threads in the warp deviate. Currently, the size of a warp is 32 threads.

- Thread: Each thread within a thread block executes an instance of the kernel and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results.

CUDA encourages users to spawn massive numbers (in the order of tens of thousands) of threads to achieve a balance between pure computation and memory load/stores, *i.e.*, to hide unnecessary latencies. Context switching between threads has nearly no cost (can be done in a single clock cycle), which in contrast to CPU threads. With multiple warps time-sharing a CUDA core and using the fast Shared memory, the utilization of GPU computation resources is much higher, putting less pressure on the memory bandwidth, which is a major bottleneck for high-performance computing applications on general-purpose processors.

1.3 Hypothesis

Recent microprocessor architectures are providing rapidly increasing amount of built-in parallelisms. Our research focuses on heterogeneous GPGPUs, which are becoming widely adopted in parallel processing. This new trend provides new opportunities to rethink the tool chain of programming including languages, programming models and compilation frameworks.

The hypothesis of this dissertation is:

Data parallelism provides the means to exploit the massive throughput of the increasing number of computation cores featured by today's microprocessors. A fundamental redesign of the tool chain including languages, programming models and compilation frameworks for data parallelism has the potential to significantly increase programmability and performance in such environments.

1.4 Organization

The remainder of this document is split into the following parts.

- We assess the benefits of exploiting the computational power of GPUs to study two fundamental problems in document mining, namely to calculate TF-IDF (Term Frequency-Inverse Document Frequency) and cluster a large set of document (Chapter 2). We transform traditional algorithms into accelerated parallel counterparts that can be efficiently executed on many-core GPU architectures. We evaluate our implementations on various platforms ranging from stand-alone GPU desktops to Beowulf-like clusters equipped with contemporary GPU cards. We observe at least one order of magnitude speedups over CPU-only desktops and clusters. This demonstrates the potential of exploiting GPU clusters to efficiently solve massive document mining problems. Such speedups combined with the scalability potential and accelerator-based parallelization are unique in the domain of document-based data mining, to the best of our knowledge.

- We propose GStream, a general-purpose, scalable data streaming framework on GPUs, to demonstrate the GPU's ability to operate on general streaming applications (Chapter 3). We provide powerful, yet concise language abstractions suitable to describe conventional algorithms as streaming problems. We then project these abstractions onto GPUs to fully exploit their inherent massive data-parallelism. By providing a transparent memory transfer interface, we show that the proposed framework provides flexibility, programmability and performance gains for various benchmarks from a collection of domains, including but not limited to data streaming, data parallel problems and numerical codes.

- We take a domain specific language approach to provide an efficient implementation of 3D iterative stencil computations on GPUs (Chapter 4). We observe that parameter tuning is necessary for heterogeneous GPUs to achieve optimal performance with respect to the parameter search

space. Our proposed framework takes a most concise specification of stencil behavior from the user as a single formula, auto-generates tunable code from it, systematically searches for the best configuration and generates the code with optimal parameter configurations for different GPUs. This auto-tuning approach guarantees adaptive performance for different generations of GPUs while greatly enhancing programmer productivity. Experimental results show that the delivered floating-point performance is very close to previous handcrafted work and outperforms other auto-tuned stencil codes by a large margin.

- In order to automatically exploit data parallelism for current GPUs, we develop a source-to-source compiler framework to convert NESL, a nested data-parallel language, into CUDA code (Chapter 5). As data-parallelism is the fundamental element in NESL, we show how natural it is to fit such languages into SIMT (single instruction multiple threads) environments. Preliminary results indicate that the resulting code still preserves performance advantages over manually written code running on the latest multi-core CPUs.

- Last but not the least, we design a new hierarchical data-parallel language called HiDP and build another source-to-source compiler that converts HiDP into efficient CUDA code (Chapter 6). The support of nested parallel map structure fits today's hierarchical microprocessors better than the segmented operations in NESL. The experimental results demonstrate that the emited code achieves performance closing to hand-coded CUDA code.

Chapter 2

Document Clustering on GPU Clusters

2.1 Introduction

Document clustering, or text clustering, is a sub-field of data clustering where a collection of documents are categorized into different subsets with respect to document similarity. Such clustering occurs without supervised information, *i.e.*, no prior knowledge of the number of resulting subsets or the size of each subset is required. Clustering analysis in general is motivated by the explosion of information accumulated in today's Internet, *i.e.*, accurate and efficient analysis of millions of documents is required within a reasonable amount of time. This trend has resulted in a myriad of clustering algorithms developed lately. A recent flocking-based algorithm [41] implements the clustering process through the simulation of mixed-species birds in nature. In this algorithm, each document is represented as a point in a two-dimensional Cartesian space. Initially set at a random coordinate, each point interacts with its neighbors according to a clustering criterion, *i.e.*, typically the similarity metric between documents. This algorithm is particularly suitable for dynamical streaming data and is able to achieve global optima, much in contrast to our algorithmic solutions [108].

In this chapter, we first solve one of the fundamental problems in document mining, namely that of calculating TF-IDF vectors of documents. The TF-IDF vector is subsequently utilized to quantify document similarity in document clustering algorithms. We show how to re-design the traditional algorithm into a CPU-GPU co-processing framework and we demonstrate up to 10X speedup over a single-node CPU desktop.

In a second step, we aim at clustering at least one million documents at a time based on the TF-IDF-like similarity metric. In document clustering, the size of each document varies and can reach up to several kilo-bytes. Therefore, document clustering imposes an even higher pressure on memory usage than traditional data mining, where data set is of much smaller and constant size. Unfortunately, many accelerators, including GPUs, do not share memory with their host systems, nor do they provide virtual memory addressing. Hence, there is no means to automatically transfer data between GPU memory and

host main memory. Instead, such memory transfers have to be invoked explicitly. The overhead of these memory transfers, even when supported by DMA, can nullify the performance benefits of execution on accelerators. Hence, a thorough design to assure well-balanced computation on accelerators and communication / memory transfer to and from the host computer is required, *i.e.*, overlap of data movement and computation is imperative for effective accelerator utilization. Moreover, the inherently quadratic computational complexity in the number of documents and the large memory footprints, however, make efficient implementation of flocking for document clustering a challenging task. Yet, the parallel nature of such a model bears the promise to exploit advances in data-parallel accelerators for distributed simulation of flocking.

As a result, we investigate the potential to purse our goal on a cluster of computers equipped with NVIDIA CUDA-enabled GPUs. We are able to cluster one million documents over sixteen NVIDIA GeForce GTX 280 cards with 1GB on-board memory each. Our implementation demonstrates its capability for weak scaling, *i.e.*, execution time remains constant as the amount of documents is increased at the same rate as GPUs are added to the processing cluster. We have also developed a functionally equivalent multi-threaded MPI application in C++ for performance comparison. The GPU cluster implementation shows dramatic speedups over the C++ implementation, ranging from 30X to more than 50X speedups.

The contributions of this work are the following:

- We design highly parallelized methods to build hash tables on GPU as a premise to calculate TF-IDF vectors for a given set of documents.
- We apply multiple-species flocking (MSF) simulation in the context of large-scale document clustering on *GPU clusters*. We show that the high I/O and computational throughput in such a cluster meets the demanding computational and I/O requirements.
- In contrast to previous work that targeted GPU clusters [48, 38], our work is one of the first to utilize CUDA-enabled GPU clusters to accelerate *massive data mining applications*, to the best of our knowledge.
- The solid speedups observed in our experiments are reported *over the entire application* (and not just by comparing kernels without considering data transfer overhead to/from accelerator). They clearly demonstrate the potential for this application domain to benefit from acceleration by GPU clusters.

The rest of the chapter is organized as follows. We begin with the background description in Section 2.2. The design and implementation of TF-IDF calculation and document clustering are presented in Section 2.3 and 2.4, respectively. In Section 2.5, we show various speedups of GPU clusters against CPU clusters in different configurations. Related work is discussed in Section 2.6 and a summary is given in Section 2.7.

2.2 Background Description

In this section, we describe the algorithmic steps of TF-IDF and document clustering, and discuss details of the target programming environments.

2.2.1 TF-IDF

Term frequency (TF) is a measure of how important a term is to a document. The ith term's tf in document j is defined as:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{2.1}$$

where $n_{i,j}$ is the number of occurrences of the term in document d_j and the denominator is the number of occurrences of all terms in document d_j.

The inverse document frequency (IDF) measures the general importance of the term in a corpus of documents. This is done by dividing the number of all documents by the number of documents containing the term and then taking the logarithm.

$$idf_i = log\frac{|D|}{|\{d_j : t_i \in d_j\}|} \tag{2.2}$$

where $|D|$ is the total number of documents in the corpus and $|\{d_j : t_i \in d_j\}|$ is the number of documents containing term t_i.

Then, the TF-IDF value of the ith term in document j is:

$$tfidf_{i,j} = tf_{i,j} * idf_i \tag{2.3}$$

The idea of TF-IDF can be extended to compare the similarities of two documents d_i and d_j. One of the simple way is to apply the similarity metric between any pair of documents i and j:

$$Sim_{i,j} = \sum_k |tfidf_{k,i} - tfidf_{k,j}|^2 \tag{2.4}$$

for k over all terms of both document i and j. Obviously, the smaller the value is, the more similar these two documents are considered.

There are many ways to calculate the TF-IDF given a corpus of documents. The most straightforward method, also used by us, is illustrated in Figure 2.1. The first step, which is part of the document preprocessing prior to the core TF-IDF calculation, excerpts and tokenizes each word of a document. It is also in this step that the stop words are removed. Stop words, also known as the noise words, are common words that do not contribute to the uniqueness of the document [3]. In the second step, some

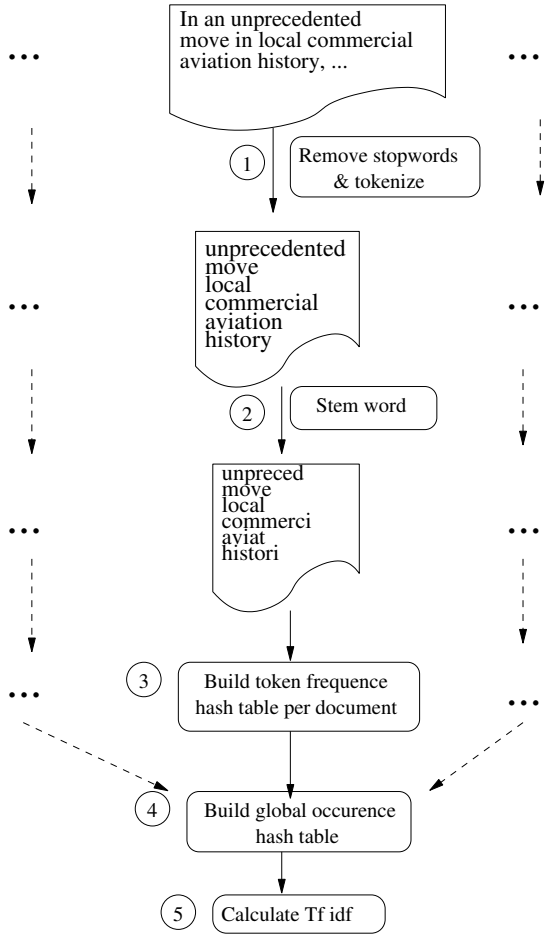


Figure 2.1: TF-IDF Workflow

cognate words are transformed into one form by applying certain stemming patterns for each. This is necessary to obtain results with higher precision [73]. In step three, the document hash table is built for each document. The <key, value> pairs in the token hash table are the unique words that appear in the document and their occurrence frequencies, respectively. In step four, all of these token hash tables are reduced into one global occurrence table in which the keys remain the same, but values represent the number of documents that contain the associated key. The TF-IDF for each term can be easily calculated by looking up the corresponding values in the hash tables according to Equation 2.3 as seen in step five.

2.2.2 Flocking-based Document Clustering

The goal of document clustering is to form groups of individuals that share certain criteria. Document similarity derived from TF-IDF provides the foundation to determine such similarities. In flocking-

based clustering, the behavior of a boid (individual) is based only on its neighbor flock mates within a certain range. Reynolds [101] describes this behavior in a set of three rules. Let $\vec{p_j}$ and $\vec{v_j}$ be the position and velocity of boid j. Given a boid noted as x, suppose we have determined N of its neighbors within radius r. The description and calculation of the force by each rule is summarized as follows:

- *Separation*: steer to avoid crowding local flock mates

$$\vec{f_{sep}} = -\sum_{i}^{N} \frac{\vec{p_x} - \vec{p_i}}{r_{i,x}^2} \tag{2.5}$$

  where $r_{i,x}$ is the distance between two boids i and x.

- *Alignment*: steer towards the average heading of local flock mates

$$\vec{f_{ali}} = \frac{\sum_i^N \vec{v_i}}{N} - \vec{v_x} \tag{2.6}$$

- *Cohesion*: steer to move toward the average position of local flock mates

$$\vec{f_{coh}} = \frac{\sum_i^N \vec{p_i}}{N} - \vec{p_x} \tag{2.7}$$

The three forces are combined to change the current velocity of the boid. In case of document clustering, we map each document as a boid that participates in flocking formation. For similar neighbor documents, all three forces are combined. For non-similar neighbor documents, only the *separation* force is applied.

2.3 Design and Implementation of TF-IDF Calculation

One of the key challenges in algorithmic design for GPGPUs is to keep all processing elements busy. NVIDIA's philosophy to ensure high utilization is to oversubscribe, *i.e.*, more parallel work is dispatched than there are physical stream processors available. Using latency-hiding techniques, a processor stalled on a memory reference can thus simply switch context to another dispatched work unit.

In order to fully utilize the large number of streaming processors in NVIDIA's GPUs, we process files in batches with the batch size chosen as 96, a heuristic number to balance the disk I/O and GPU processing time. Several kernels are developed to implement the steps described in Section 2.2.1. Each batch process requires extensive data movement between host and GPU memories by DMA. First, to handle a large amount of documents/files, especially when total document size is larger than the GPU global memory, the document hash tables needs to be flushed out to host memory once they are completely constructed. Second, the raw data of a document is pushed from host memory to GPU global

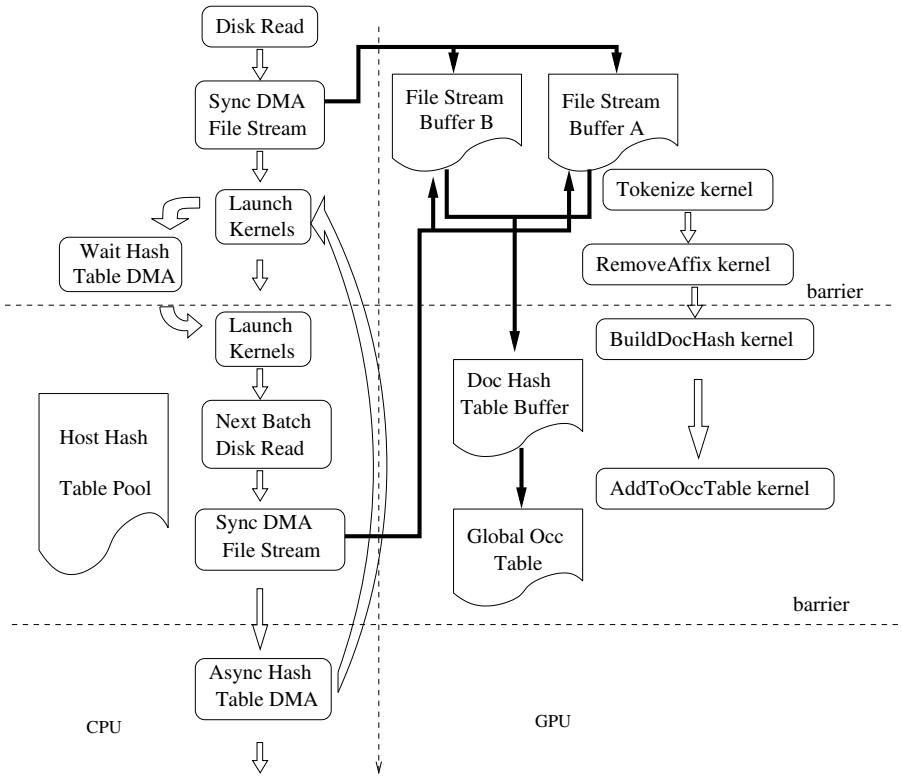


Figure 2.2: CPU/GPU Collaboration Framework

memory at the beginning of each batch process. To reduce the overhead of memory movement, we developed the CPU/GPU collaboration framework shown in Figure 2.2.

In each batch iteration, the CPU thread first launches the two preprocessing kernels (Tokenize_kernel and RemoveAffix_kernel) asynchronously. Before invoking the next kernels (BuildDocHash_kernel and AddToOccTable_kernel) that write to the document and global occurrence hash table buffers in the GPU's global memory, it waits for the completion signal of the previous issued DMA. This DMA saves the document hash tables in the previous batch to host memory. When the GPU is busy generating the document hash tables and inserting tokens into the global occurrence table, the CPU can prefetch the next batch of files from disk and copy them to an alternate file stream buffer. At the end of the batch iteration, the CPU again asynchronously issues a memory copy of the document hash table to the host's memory. Only in the next batch's iteration will the completion of this DMA be synchronized. In this manner, part of the DMA time is overlapped with the GPU calculation by (a) double buffering the document raw data in GPU and (b) overlapping the hash table memory copy in the current batch with the stream preprocessing (tokenize and stem kernels) of the next batch [36].

To further reduce the DMA overhead, one may reduce the size of the document hash table. This

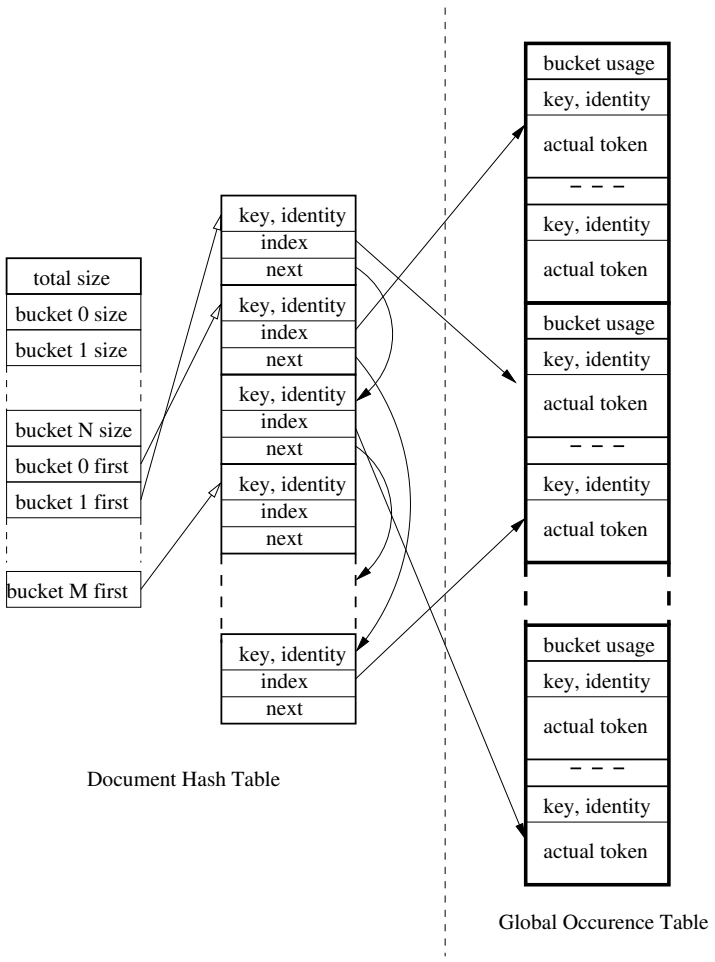


Figure 2.3: Hash Table Data Structures

table differs from the global occurrence table, which resides in GPU global memory but need not be copied to host until the end of execution. Therefore, the data structures of these tables differ slightly as shown in Figure 2.3. Since no hash insertion or deletion operations will be performed afterwards, we store this table as a linked list. The data structure contains a header and an array of entries, which are stored continuously if they belong to the same bucket. The header is used to determine the bucket size and to find the first entry in each bucket. In contrast, the global hash table consists of a big array of entries evenly divided into buckets. Because the number of unique terms is considered limited no matter how large the corpus size is, the number of buckets and the bucket size can be chosen sufficiently large to avoid possible bucket overflows.

Another effort to reduce the size of the document hash table avoids storing the actual term/word in the table. Instead, every entry simply maintains an index pointing to the corresponding entry in the

global occurrence table where the actual term is saved. To reduce the number of hash key computations at hash insertion and during hash searches, the key is saved as an "unsigned long" in both hash tables. To further reduce the probability of hash collisions (two terms sharing the same key), another field called identity is added as an "unsigned int" to help differentiate terms. The identity is then constructed as $(term\ length << 16)|(first\ char << 8)|(last\ char)$.

Upon investigation, we determined that atomic operations supported by certain GPUs via CUDA are facilitating the construction of a concise document hash table without adversely affecting the parallelism of the algorithm. We alternatively provide another method to generate the same hash table for GPUs without support for atomic operations. Even though the latter method is slower than the first, it is required for GPU devices that do not have atomic operation support (*i.e.*, devices with CUDA compute capability 1.0 or earlier).

2.3.1 Hash Table Updates using Atomic Operations

Access to hash table entries *via* atomic operations is realized in two steps as depicted in Figure 2.4. In the first step, the document stream is evenly distributed to a set of CUDA threads. The number of threads, L, is chosen explicitly to maximize GPU's utilization. A buffer storing the intermediate hash table, which is close to the structural layout of the global occurrence table, but with a smaller number of buckets K, is used to sort terms by their bucket IDs. Every time a thread encounters a new term in the stream and obtains its bucket ID, it issues an atomic increment (atomic-add-one) operation to affect the bucket size. (Notice that the objective of this algorithmic TF-IDF variant is not to identify identical terms. Instead, its chief objective is to compute a similarity metric.) If we assume that terms are distributed randomly, then contention during the atomic increment operation is the exception, *i.e.*, threads of the same warp are likely atomically incrementing disjoint bucket size entries.

In the next step, the intermediate hash table is reduced to the final, more concise document hash table shown in Figure 2.3. Each CUDA thread traverses one bucket in the intermediate hash table, detects duplicate terms, and, if finds a new term, reserves a place in the entry array by atomically incrementing the total size. It then pushes the new entry into the header of the linked bucket list. Since different threads operate on disjoint buckets, each linked list per bucket is accessed in mutual exclusion, which guarantees absence of write conflicts between threads.

2.3.2 Hash Table Updates without Atomic Operations

In GPUs without atomic instruction support, the document stream is first split into M packets, each of which is pushed into a different hash sub-table owned by one thread in a block, as shown in step 1 of Figure 2.5. By giving each thread a separate hash sub-table, we guarantee write protection (mutually exclusive writes of the values) between threads. In step 2, K threads are re-assigned to different buckets of the sub-table, identical terms are found in this step, and statistics for each bucket are generated.

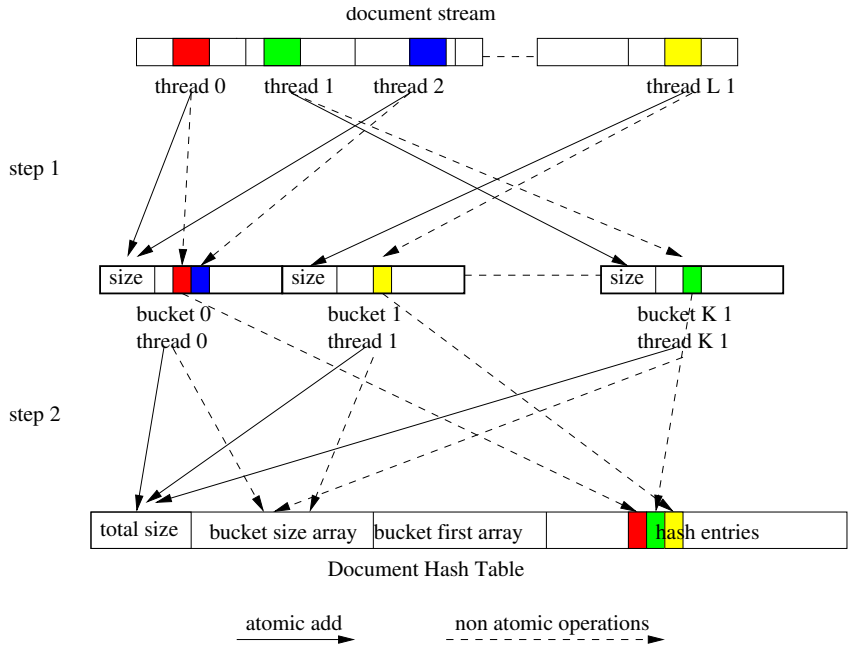


Figure 2.4: Building a Hash Table with Atomic Operations

Because terms have been grouped by their keys in step 1, there will be no write conflicts between threads at this step either. The bucket size information is processed in step 3 to finally merge sub-tables to compose the final document hash table.

2.3.3 Discussions

The two procedures detailed above to handle hash tokens in a document do not require information from any other documents. Thus, each document can be processed simultaneously and independently in different GPU blocks. With a sufficiently large number of documents, we can fully utilize the GPU cores and exploit NVIDIA's latency hiding on memory references through oversubscription. However, in the first step of the second method, the number of packets M per document is delimited due to memory constraint and the efficiency of the following steps. We choose a value of $M = 16$ in our implementation. To compensate for this constraint, we can spawn more threads L in the first method, *e.g.*, by choosing $L = 512$. This constraint on parallelism results in a non-atomic approach that is slower than its atomic variant.

From the memory usage's perspective, the non-atomic approach consumes more global memory simply because the intermediate hash tables in the non-atomic approach are larger than that in the atomic approach. Both of the above methods cannot handle very large single documents that exceed the size of the global memory. Since our problem domain is that of Internet news articles, which typically do

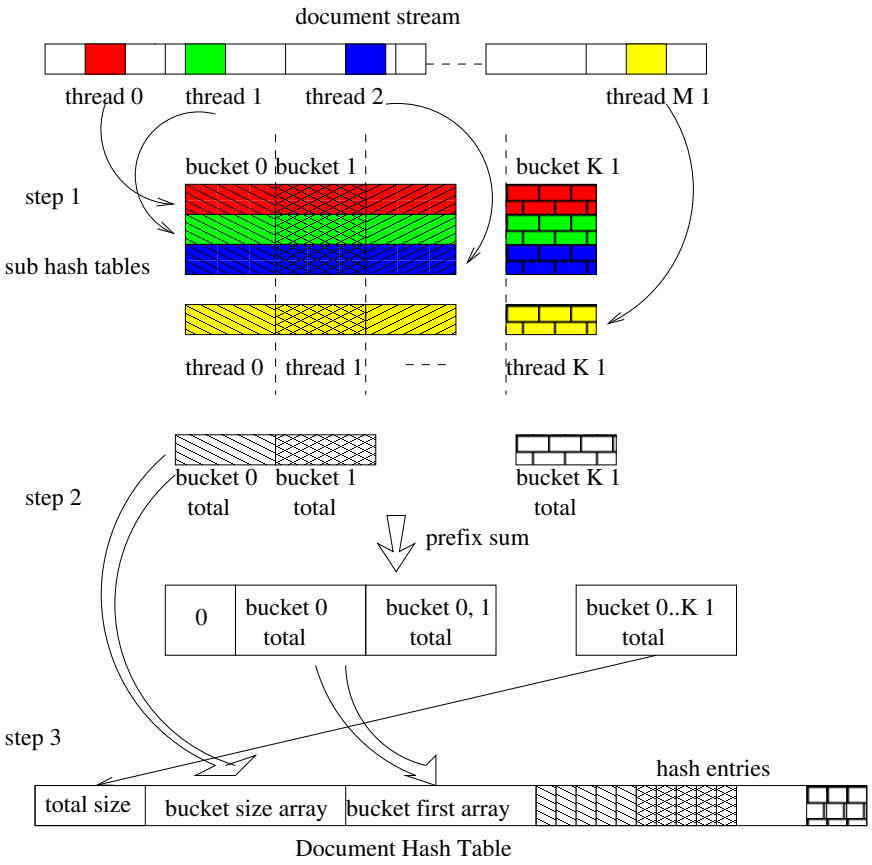


Figure 2.5: Building a Hash Table without Atomic Operation

not exceed more than 10K words, documents fits in memory for our implementation. This framework is even suitable for arbitrarily large corpus sizes as we could reused without changes both intermediate hash tables and the document hash table, the latter of which is flushed to host memory for each batch of files.

2.4 Design and Implementation of Document Clustering

2.4.1 Programming Model for Data-parallel Clusters

We have developed a programming model targeted at message passing for CUDA-enabled nodes. The environment is motivated by two problems that surface when explicitly programming with MPI and CUDA abstraction in combination:

- Hierarchical memory allocation and management have to be performed manually, which often burdens programmers.

- Sharing one GPU card among multiple CPU threads can improve the GPU utilization rate. However, explicit multi-threaded programming not only complicates the code, but may also result in inflexible designs, increased complexity and potentially more programming pitfalls in terms of correctness and efficiency.

To address these problems, we have devised a programming model that abstracts from CPU/GPU co-processing and mitigates the burden of the programmer to explicitly program data movement across nodes, host memories and device memories. We next provide a brief summary of the key contributions of our programming model (see [124] for a more detailed assessment):

- We have designed a *distributed object interface* to unify CUDA memory management and explicit message passing routines. The interface enforces programmers to view the application from a data-centric perspective instead of a task-centric view. To fully exploit the performance potential of GPUs, the underlying run-time system can detect data sharing within the same GPU. Therefore, the network pressure can be reduced.

- Our model provides the means to spawn a flexible number of host threads for parallelization that may *exceed* the number of GPUs in the system. Multiple host threads can be automatically assigned to the same MPI process. They subsequently share one GPU device, which may result in higher utilization rate than single-threaded host control of a GPU. In applications where CPUs and GPUs co-process a task and a CPU cannot continuously feed enough work to a GPU, this sharing mechanism utilizes GPU resources more efficiently.

- An interface for advanced users to control thread scheduling in clusters is provided. This interface is motivated by the fact that the mapping of multiple threads to physical nodes affects performance depending on the application's communication patterns. Predefined communication patterns can simply be selected so that communication endpoints are automatically generated. More complex patterns can be supported through reusable plug-ins as an extensible means for communication.

We have designed and implemented the flocking-based document clustering algorithm in GPU clusters based on this GPU cluster programming model. In the following, we discuss several application-specific issues that arise in our design and implementation.

2.4.2 Preprocessing

The prerequisite of document clustering is to have a standard means to measure similarities between any two documents. While the TF-IDF concept exactly matches this need, there are two practical issues when targeting clusters:

- There is a reduce step (step 4 in Figure 2.1) to generate a single global occurrence hash table. This is a high payload all-to-all communication in clusters and thus is not scalable.

- The TF-IDF calculation cannot start until all documents have been processed and inserted in the global occurrence table. Therefore, it is not suited for stream processing.

A new term weighting scheme called term frequency-inverse corpus frequency (TF-ICF) has been proposed to solve the above problems at the scale of massive amounts of documents [100]. It does not require term frequency information from other documents within the processed document collections. Instead, it pre-builds the ICF table by sampling a large amount of existing literature off-line. Selection of corpus documents for this training set is critical as similarities between documents of a later test set are only reliable if both training and test sets share a common base dictionary of terms (words) with a similar frequency distribution of terms over documents. Once the ICF table is constructed, ICF values can be looked up very efficiently for each term in documents while TF-IDF would require dynamic calculation of these values. The TF-ICF approach enables us thus to generate document vectors in linear time.

2.4.3 Flocking Space Partition

The core of the flocking simulation is the task of neighborhood detection. A sequential implementation of the detection algorithm has $O(N^2)$ complexity due to pair-wise checking of N documents. This simplistic design can be improved through space filtering, which prunes the search space for pairs of points whose distances exceed a threshold.

One way to split the work into different computational resource is to assign a fixed number of documents to each available node. Suppose there are N documents and P nodes. In every iteration of the neighborhood detection algorithm, the positions of local documents are broadcast to all other nodes. Such partitioning results in a lower communication overhead proportional to the number of nodes, and the detection complexity is reduced linearly by P per node for a resulting overhead of $O(N^2/P)$.

Instead of partitioning the documents in this manner, we break the virtual simulation space into row-wise slices. Each node handles just those documents located in the current slice. Broadcast messages that are previously required are replaced by point-to-point messages in this case. This partitioning is illustrated in Figure 2.6. After document positions are updated in each iteration, additional steps are performed to divide all documents into three categories. *Migrating documents* are those that have moved to a neighbor slice. *Neighbor documents* are those that are on the margin of the current slice. In other words, they are within the range of the radius r of neighbor slices. All other are *internal documents* in the sense that they do not have any effects on the documents in other nodes. Since the velocity of documents is capped by a maximal value, it is impossible for the migrating documents to cross an entire slice in one timestep. Both the migrating documents and neighbor documents are transferred to neighbor slices at the beginning of the next iteration. Since the neighborhood radius r is much smaller than the virtual space's dimension, the number of migrating documents and neighbor documents are expected to be much smaller than that of the internal documents.

Sliced space partitioning not only splits the work nearly evenly among computing nodes but also reduces the algorithmic complexity in sequential programs. Neighborhood checks across different nodes are only required for neighbor documents within the boundaries, not for internal documents. Therefore, on average, the detection complexity on each node reduces to $O(N^2/P^2)$ for slides partitioning, which is superior to traditional partitioning with $O(N^2/P)$.

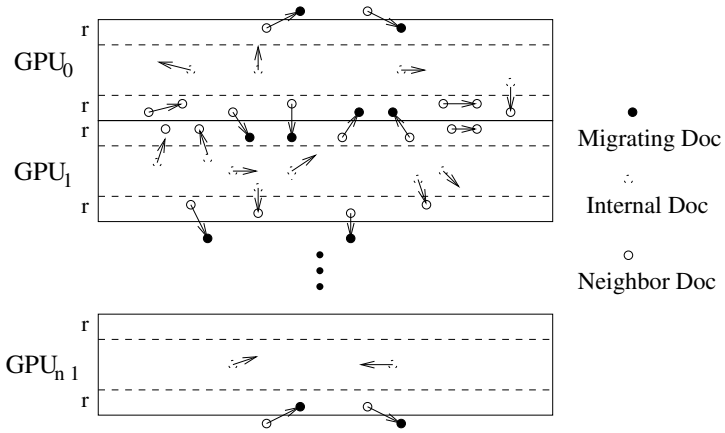


Figure 2.6: Simulation Space Partition

2.4.4 Document Vectors

An additional benefit of MSF simulation is the similarity calculation between two neighbor documents. Similarities could be pre-calculated between all pairs and stored in a triangular matrix. However, this is infeasible for very large N because of a space complexity of $O(N^2/2)$, which dauntingly exceeds the address space of any node as N approaches a million. Furthermore, devising an efficient partition scheme to store the matrix among nodes is difficult due to the randomness of similarity look-ups between any pair of nearby documents. Therefore, we devote one kernel function to calculating similarities in each iteration. This results in some duplicated computations, but this method tends to minimize the memory pressure per node.

The data required to calculate similarities is a document vector consisting of an index of each unique word in the TF-ICF table and its associated TF-ICF values. To compute the similarity between two documents, as shown in Equation (2.4), we need a fast method to determine if a document contains a word given the word's TF-ICF index. Moreover, the fact that we need to move the document vector between neighbor nodes also requires that the size of the vector should be kept small.

The approach we take is to store document vectors in an array sorted by the index of each unique

word in the TF-ICF table. This data structure combines the minimal memory usage with a fast parallel searching algorithm. Riech [103] describes an efficient algorithm to calculate the Euclidean similarities between any two sorted arrays. But this algorithm is iterative in nature and not suitable for parallel processing.

We develop an efficient CUDA kernel to calculate the similarity of two documents given their sorted document vectors as shown in Algorithm 1. The parallel granularity is set so that each block takes one pair of documents. Document vectors are split evenly by threads in the block. For each assigned TF-ICF value, each thread determines if the other document vector contains the entry with the same index. Since the vectors are sorted, a binary search is conducted to lower the algorithmic complexity logarithmic time. A reduction is performed at the end to accumulate differences.

2.4.5 Message Data Structure

In sliced space partitioning, each slice is responsible to generate two sets of messages for the slices above and below. The corresponding message data structures are illustrated in Figure 2.7. The document array contains a header that enumerates the number of neighbors and migrating documents in the current slice. Their global indexes, positions and velocities are stored in the following array for neighborhood detection in a different slice. Due to the various sizes of each document's TF-ICF vector and the necessity to minimize the message size, we concatenate all vectors in a vector array without any padding. The offset of each vector array is stored in a metadata offset array for fast access. This design offers efficient parallel access to each document's information.

2.4.6 Optimizations

The algorithmic complexity of sliced partitioning decreases quadratically with the number of partitions (see Section 2.4.3). For a system with a fixed number of nodes, a reduction in complexity could be achieved by exploiting multi-threading within each node. However, in practice, overhead increases as the number of partitions become larger. This is particularly this case for communication overhead. As we will see in Section 2.5, the effectiveness of such performance improvements differs from one system to another.

At the beginning of each iteration, each thread issues two non-blocking messages to its neighbors to obtain the neighboring and migrating documents' statuses (positions) and their vectors. This is followed by a neighbor detection function that searches its neighbor documents within a certain range for each internal document and migrated document. The search space includes every internal, neighbor and migrating document. We can split this function into three sub-functions: (a) internal-to-internal document detection; (b) internal-to-neighbor/migrating document detection and (c) migrating-to-all document detection. Sub-function (a) does not require information from other nodes. We can issue this kernel in parallel with communication. Since the number of internal documents is much larger than neighbor and

Algo 1: Document Vector Similarity (CUDA Kernel)

```
// calculate the similarities between two DocVecs
__device__ void docVecSimilarity(DocVec* lhs, DocVec *rhs, float *output) {
  float sim(0.0f);
  float commonSim(0.0f);
  for (int i = 0; i < lhs→NumEntries; i += blockIdx.x) {
      float tficf = biSearch(entry, rhs→vectors);
      sum += pow(entry→tficf − tficf, 2);
      commonSim += pow(tficf, 2);
  }
  // ... reduce to threadIdx.x(0), store in sum
  __syncthreads();
  if (threadIdx.x == 0) {
      sum −= commonSim;
      sum = sqrtf(sum);
      // write to global memory
      *output = sum;
  }
}

__device__ float biSearch(VecEntry *entry, DocVector *vector) {
  int idx = entry→index;
  int leftIndex = 0;
  int rightIndex = vector→NumEntries;
  int midIndex = vector→NumEntries/2;
  while(true) {
      int docIdx;
      docIdx = vector→vectors[midIndex].index;
      if (docIdx < idx)
        leftIndex = midIndex + 1;
      else if (docIdx > idx)
        rightIndex = midIndex − 1;
      else
        break;

      if (leftIndex > rightIndex)
        return 0.0f;
      midIndex = (leftIndex + rightIndex)/2;
  }
  return vector→vectors[midIndex].tficf;
}
```

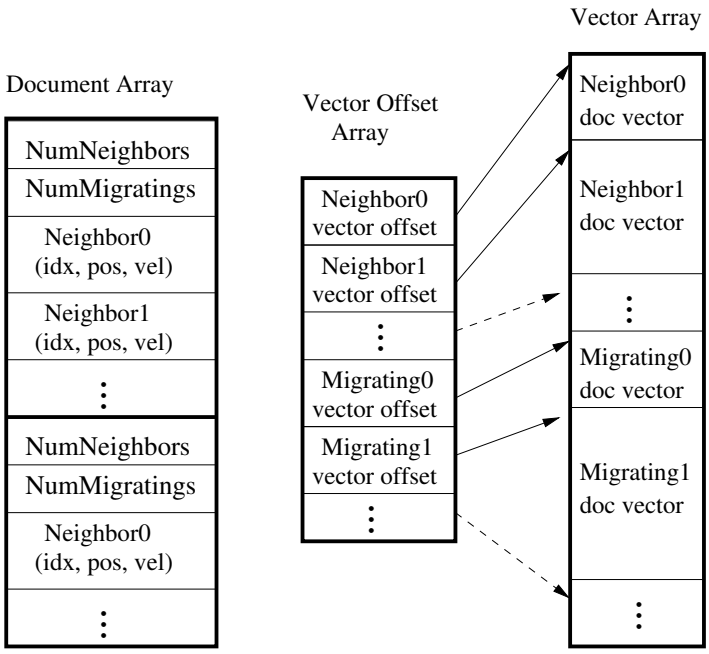


Figure 2.7: Message Data Structures

migrated documents, we expect the execution time for sub-function (a) to be much larger than that of (b) or (c). From the system's point of view, either the communication or neighbor detection functions affects the overall performance.

One of the problems in simulating massive documents via the flocking-based algorithm is that as the virtual space size increases, the probability of flock formation diminishes as similar groups are less likely to meet each. In nature-inspired flocking, no explicit effort is made within simulations to combine similar species into a unique group. However, in document clustering, we need to make sure each cluster has formed only *one group* in the virtual space in the end without flock intersection. We found that an increase in the number of iterations helps in achieving this objective. We also dynamically reduce the size of the virtual space throughout the simulation. This increases the likelihood of similar groups to merge when they become neighbors.

2.4.7 Work Flow

The work flow for each space partition at an iteration is shown in Figure 2.8. Each thread starts by issuing asynchronous messages to fetch information from neighboring threads. Messages include data such as positions of the documents that have migrated to the current thread and documents at the margin of the neighbor slices. Those documents' TF-ICF vectors are encapsulated in the message for similarity calculation purposes, as discussed later.

Internal-to-internal document detection can be performed in parallel with message passing (see Section 2.4.6). The other two detection routines, in contrast, are serialized with respect to message exchanges. Once all neighborhoods are detected, we calculate the similarities between the documents belonging to the current thread and their detected neighbors. These similarity metrics are utilized to update the document positions in the next step where the flocking rules are applied.

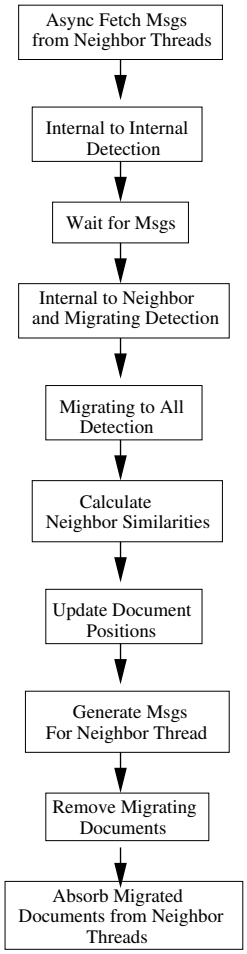


Figure 2.8: Work Flow for a Thread in Each Iteration

Once the positions of all documents have been updated, some documents may have moved out the boundary of the current partition. These documents are removed from the current document array and form the messages for neighboring threads for the next iteration. Similarly, migrated documents received through messages from neighbors are appended to the current document array. This post-processing is

performed in the last three steps in Figure 2.8.

Table 2.1: Experiment Platforms

	16 GPUs (NCSU)	16 CPUs (NCSU)	3 GPUs (ORNL)	3 CPUs (ORNL)
Nodes	16	16	4	4
CPU Cores	AMD Athlon Dual	AMD Athlon Dual	Intel Quad Q6700	Intel Quad Q6700
CPU Frequency	2.0 GHz	2.0 GHz	2.67 GHz	2.67 GHz
System Memory	1 GB	1 GB	4 GB	4 GB
GPU	16 GTX 280s	Disabled	3 Tesla C1060	Disabled
GPU Memory	1 GB	N/A	4 GB	N/A
Network	1 Gbps	1 Gbps	1 Gbps	1 Gbps

2.5 Experimental Results

2.5.1 Experiment Setups

We conduct two independent sets of experiments to show the performance of our TF-IDF and document clustering results.

TF-IDF experiments are conducted on a stand-alone desktop in two configurations: with GPU enabled and disabled. When the GPU is disabled, we assess the performance of a functionally equivalent CPU baseline version (single-threaded in C/C++). The test platform utilizes Fedora 8 Core Linux with a dual-core AMD Athlon 2 GHz CPU with 2 GB of memory. The installation includes the CUDA 2.0 beta release and NVIDIA's Geforce GTX 280 as GPU devices. The test input data is selected from Internet news documents with variable sizes ranging from around 50 to 1000 English words (after stop-word removal). The average number of unique word in each article is about 400 words.

Similarly, the document clustering experiments are conducted on GPU-accelerated clusters with GPUs enabled and disabled. In the absence of GPUs, the performance of a multi-threaded CPU version of the clustering algorithm is assessed. In this version, internal document vectors are stored in STL hash containers instead of sorted document vectors used in GPU clusters. This combines benefits of fast serial similarity checking with ease of programming. The message structure is the same in both implementations. Hence, functions are provided to convert STL hashes to vector arrays and vice versa. In document clustering experiments, both GPU and CPU implementations incorporate the same MPI library (MPICH 1.2.7p1 release) for message passing and the C++ boost library (1.38.0 release) for multi-threading in a single MPI process. The GPU version uses the CUDA 2.1 release.

2.5.2 TF-IDF Experiments

In TF-IDF experiments, we first compare the execution time for one batch of 96 files. The individual module speedup and their percentages in total are shown in Figure 2.9 and Figure 2.10.

Notice that the speedup on the y-axis of Figure 2.9 is depicted on a logarithmic scale. Compared to the CPU baseline implementation, we achieve more significant speedups for those modules engaged in the preprocessing phase (factor of 30 times faster in tokenize and 20 times faster in strip affixes kernels) than for those at the hash table construction phase (around 3 times faster in both document hash table and occurrence table insertion kernels). The limits in speedup during the latter are due to the multi-step hash table construction algorithms described in Section 2.3. The algorithm has certain overheads that the CPU benchmark does not contain. These overheads include (a) the construction of intermediate or hash sub-tables; (b) branching penalties suffered from the SIMD nature of GPU cores due to the imbalance in the distribution of tokens for a hash table's buckets; and (c) non-coalesced global memory access patterns as a result of the randomness of the hash key generation. Furthermore, the kernel for occurrence table insertion does not fully exploit all GPU cores because insertion is inherently serialized over files to avoid write conflicts within the same hash table bucket.

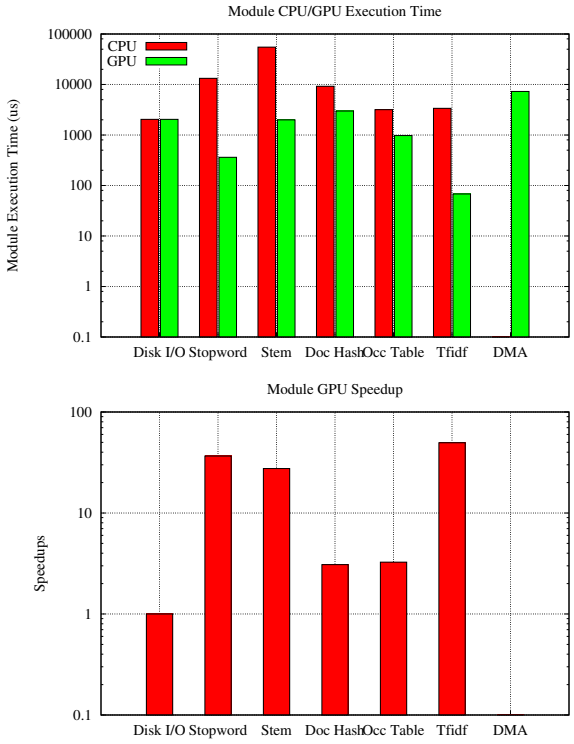


Figure 2.9: Per-Module Performance: CPU baseline vs. CUDA

We also observe a reduction in the calculation time to the extent that the DMA overhead has become the largest contributor to overall time in a *single batch* scenario accounting for almost half of the total execution time. The combined time with disk I/O exceeds the total kernel execution time on GPU.

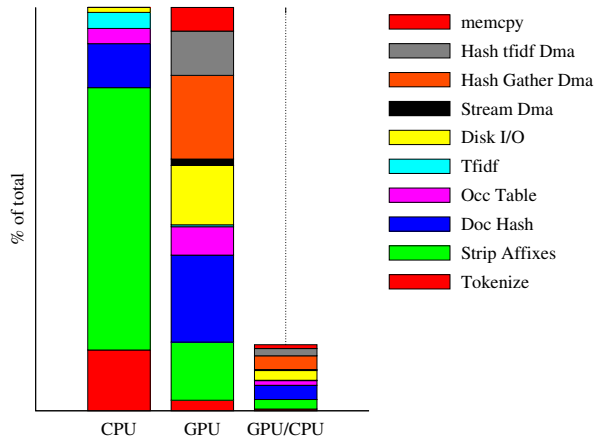


Figure 2.10: Per-Module Contribution to Overall Execution Time

The observation above gives us the motivation to mitigate the memory overhead by double buffering the stream and hash tables when the corpus size gets larger. While we cannot hide the DMA overhead of a first batch, the DMA time of subsequent batches can be completely overlapped with the computational kernels in a *multi-batch* scenario. Figure 2.11 shows the execution time of CPU and CUDA with different corpus sizes.

The execution time of the two methods (both with and with the use of atomic instructions) are measured. With almost perfect parallelization between GPU calculation and data migration, we can hide almost all the kernel execution time in the DMA transfer and disk I/O time, which indicates a lower bound of the execution time. As a result the the asymptotic average batch processing time is almost half comparing to the single batch execution time, in which case the calculation and DMA cannot be overlapped. We also observe that the overall acceleration rates are 9.15 and 7.20 times faster than the CPU baseline.

2.5.3 Flocking Behavior Visualization

We have implemented support to visualize the flocking behavior of our algorithm off-line once the positions of documents are saved after an iteration. The evolution of flocks can be seen in the three snapshots of the virtual plane in Figure 2.12, which shows a total of 20,000 documents clustered on

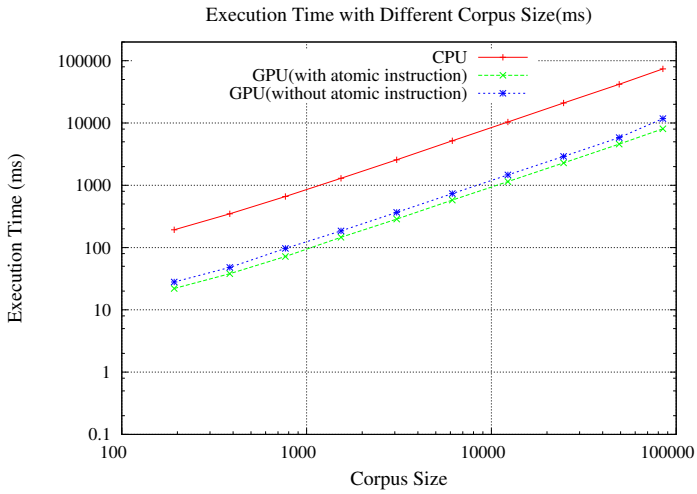
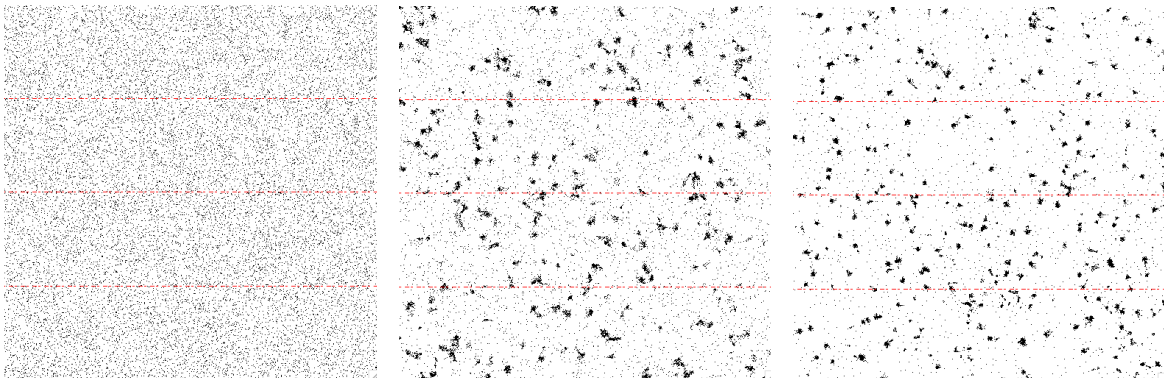


Figure 2.11: Execution Time with Different Corpus Size

four GPUs. Initially, documents are assigned at random coordinates in the virtual plane. After only 50 iterations, we observe an initial aggregation tendency. We also observe that the number of non-attached documents tends to decrease as the number of iterations increases. In our experiments, we observe that 500 iterations suffice to reach a stable state even for as many as a million documents. Therefore, we use 500 iterations throughout the rest of our experiments.



(a) Initial State (b) At Iteration 50 (c) At Iteration 500

Figure 2.12: Clustering 20K Documents in 4 GPUs

As Figure 2.12 shows, the final number of clusters in this example is quite large. This is because our input documents from the Internet cover widely divergent news topics. The resulting number is also a

factor of the similarity threshold used throughout the simulation. The smaller the threshold is / the more strict the similarity check is, the more groups we will be formed through flocking.

2.5.4 Document Clustering Performance

We first compare the performance of individual kernels on an NVIDIA GTX 280 GPU hosted on a AMD Athlon 2 GHz Dual Core PC. We focus on two of the most time-consuming kernels: detecting neighbor documents (detection for short) and neighbor document similarity calculation (similarity for short). Only the GPU kernel is measured in this step. The execution time is averaged over 10 independent runs. Each run measures the first clustering step (first iteration in terms of Figure 2.12) to determine the speedup over the CPU version starting from the initial state. The speedup at different document sizes is shown in Figure 2.13. We can see that the similarity kernel on the GPU is about 45 times faster than on a CPU at almost all document sizes. For the detection kernel, the GPU is fully utilized once the document size exceeds 20,000, which gives a raw speedup of over 300X.

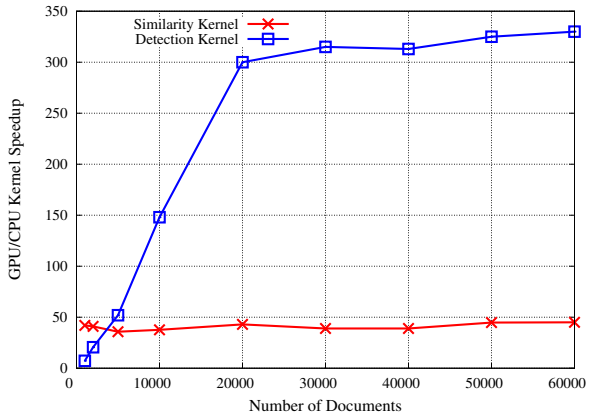


Figure 2.13: Speedups for Similarity and Detection Kernels

We next conducted experiments on two clusters located at NCSU and ORNL. On both clusters, we conducted test with and without GPUs enabled (see hardware configurations in Table 2.1). The NCSU cluster consists of sixteen nodes with CPUs and GPUs of lower RAM capacity for both CPU and GPU, while the ORNL cluster consists of fewer nodes with larger RAM capacity. As mentioned in Section 2.4.1, our programming model supports a flexible number of CPU threads that may exceed the number of GPUs on our platform. Thus, multiple CPU threads may share one GPU. In our experiments, we assessed the performance for both one and two CPU threads per GPU.

Figure 2.14 depicts the results for wall-clock time on the NCSU cluster. The curve is averaged

over the execution for both one and two CPU threads per GPU. The error bar shows the actual execution time: the maximum/minimum represent one/two CPU threads per GPU, respectively. With increasing of number of nodes, execution time decreases and the maximal number of documents that can be processed at a time increases. With 16 GTX 280s, we are able to cluster one million documents within twelve minutes. The relative speedup of the GPU cluster over the CPU cluster ranges from 30X to 50X. As mentioned in Section 2.4.6, changing the number of threads sharing one GPU may cause a number of conflicts in resource. The benefit of multi-threading in this cluster is only moderate with only up to a 10% performance gain.

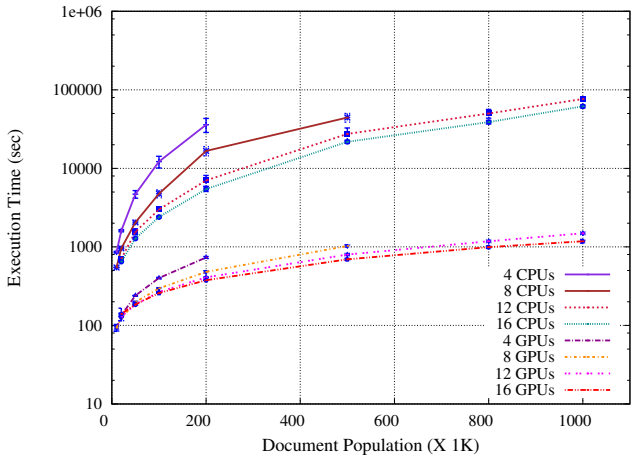


Figure 2.14: Execution Time on GTX 280 GPUs

Though the ORNL cluster contains fewer nodes, its single-GPU memory size is four times larger than that of the NCSU GPUs. This enables us to cluster one million documents with only three high-end GPUs. The execution time is shown in Figure 2.15. The performance improvement resulting for two CPU threads per GPU is more obvious in this case: at one million documents, three nodes with two CPU threads per GPU run 20% faster than the equivalent with just one CPU thread per GPU. This follows the intuition that faster CPUs can feed more work via DMA to GPUs.

Speedups on the GPU cluster for different number of nodes and documents are shown in the 3D surface graph Figure 2.16 for the NCSU cluster. At small document scale (up to 200k documents), 4 GPUs achieve the best speedup (over 40X). Due to the memory constraints in these GPUs, only 200k documents can be clustered on 4 GPUs. Therefore, speedups at 500k documents are not available for 4 GPUs. For 8 GPUs, clustering with 500k documents shows an increased performance. This surface graph illustrates the overall trends: For fewer nodes (and GPUs), speedups increase rapidly over for smaller number of documents. As the number of documents increases, speedups are initially on a plane

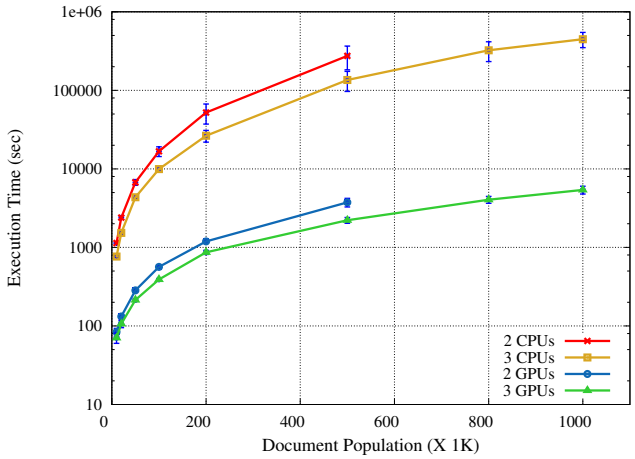


Figure 2.15: Execution Time on Tesla C1060 GPUs

with a lower gradient before increasing rapidly, *e.g.*, between 200k and 500k documents for 16 nodes (GPUs).

We next study the effect of utilizing point-to-point messages for our simulation algorithm. Because messages are exchanged in parallel with the neighborhood detection kernel for internal documents, the effect of communication is determined by the ratio between message passing time and kernel execution time: If the former is less than the latter, then communication is completely hidden (overlapped) by computation. In an experiment, we set the number of documents to 200k and vary the number of nodes from 4 to 16. We assess the execution time per iteration by averaging the communication time and kernel time among all nodes. The result is shown in Figure 2.17. For the GPU cluster, kernel execution time is always less than the message passing time. For the CPU cluster, the opposite is the case.

Table 2.2: Fraction of Communication in GPU and CPU Clusters (GPU/CPU) [in %]

Docs(k)	5	10	20	50	100	200	500	800	1000
4 nodes	74/9	67/8	64/5	58/3	52/1.5	49/0.9	NA	NA	NA
8 nodes	67/12	71/11	65/8	68/6	62/3.5	56/2	52/1.2	NA	NA
12 nodes	67/17	69/12	68/10	71/8	68/6	63/3	57/1.4	54/1.2	NA
16 nodes	63/18	63/13	71/12	69/9	65/7	66/4.2	59/1.9	60/1.5	55/1.1

Notice that the communication time for the GPU cluster in this graph includes the DMA duration for data transfers between GPU memory and host memory. The DMA time is almost two orders of magnitude less than that of message passing. Thus, the GPU communication/DMA curve almost coincides

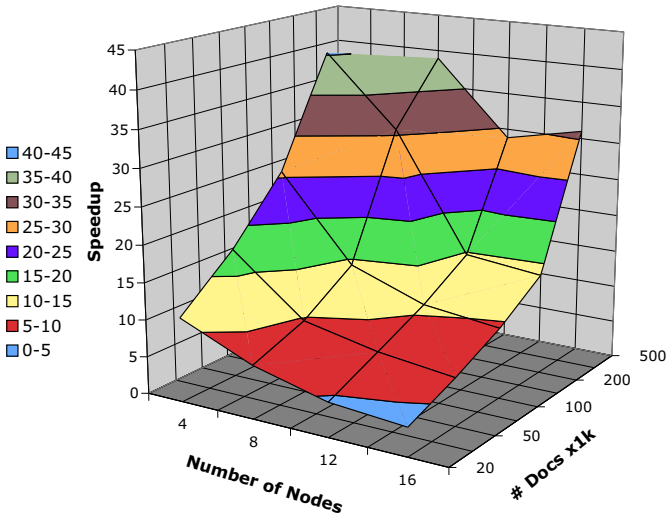


Figure 2.16: Speedups on NCSU cluster

with that of CPU cluster's communication time, even though the latter only covers pure network time as no host/device DMA is required. This implies that internal PCI-E memory bus is not a bottleneck for GPU clusters in our experiments, which is important for performance tuning efforts. The causes for this finding are: (a) Network bandwidth is much lower than PCI-E memory bus bandwidth; and (b) messages are exchanges at roughly the same time on every node at each iteration, which may cause network congestion.

We further aggregate the time spent on message passing and divide the overall sum by the total execution time to yield the percentage of time spent on communication. For CPUs, the communication time consists of only the message passing time over the network. For GPUs, the communication time also includes the time to DMA messages to/from GPU global memory over the PCI-E memory bus. Table 2.2 shows the results for both GPU and CPU clusters. Generally speaking, in both cases, the ratio of communication to computation decreases as the number of documents per thread increases. The raw kernel speedup provided by GPU has dramatically increased the communication percentage. This analysis, indicating communication as a new key component for GPU clusters while CPUs are dominated by computation, implies disjoint optimization paths: faster network interconnects would significantly benefit GPU clusters while optimizing kernels even further would more significantly benefit CPU clusters.

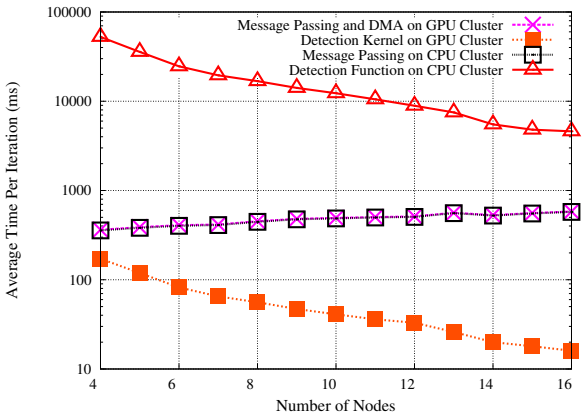


Figure 2.17: Communication and Computation in Parallel

2.6 Related Work

Our acceleration approach over CUDA to calculate document-level TF-IDF values uncovers yet another area of potential for GPUs where they outperform general-purpose CPUs. While it has been demonstrated that CUDA can significantly speedup many computationally intensive applications from domains such as scientific computation, physics and molecular dynamics simulation, imaging and the finance sector [51, 92, 105, 42, 6, 63], acceleration is less commonly used in other domains, especially those with integer-centric workloads, with few exceptions[57, 58]. This is partly due to the perception that fast (vector) floating-point calculation are the major contributor to performance benefits of GPUs. However, careful parallel algorithmic design may results in significant benefits as well. This is the premise of our work for text search workload deployment on GPUs.

Related research to document clustering can be divided into two categories: (1) fast simulation of group behavior and (2) GPU-accelerated implementations of document clustering.

The first basic flocking model was devised by Reynolds [102]. Here, each individual is referred as a "boid". Three rules are quantified to aid the simulation of flocks: separation, alignment and cohesion. Since document clustering groups documents in different subsets, a multiple-species flocking (MSF) model is developed by Cui *et al.* [41]. This model adds a similarity check to apply only the separation rule to non-similar boids. A similar algorithm is found by Momen *et al.* [88] with many parameter tuning options. Computation time becomes a concern as the need to simulate large numbers of individuals prevails. Zhou *et al.* [127] describe a way to parallelize the simulation of group behavior. The simulation space is dynamically partitioned into P divisions, where P is the number of available computing nodes. A mapping of the flocking behavioral model onto streaming-based GPUs is presented by Erra *et al.* [47] with the objective of obstacle avoidance. This study predates the most recent language/run-time support for general-purpose GPU programming, such as CUDA, which allows simulations at much larger scale.

Recently, data-parallel co-processors have been utilized to accelerate many computing problems, including some in the domain of massive data clustering. One successful acceleration platform is that of Graphic Processing Units (GPUs). Parallel *data* mining on a GPU was assessed early on by Che *et al.* [34], Fang *et al.* [49] and Wu *et al.* [119]. These approaches rely on k-means to cluster a large space of data points. Since the size of a single point is small (*e.g.*, a constant-sized vector of floating point numbers to represent criteria such as similarity in our case), memory requirements are linear to the size of individuals (data points), which is constrained by the local memory of a single GPU in practice. Previous research has demonstrated more than five times speedups using a single GPU card over a single-node desktop for several thousands documents [32]. This testifies to the benefits of GPU architectures for highly parallel, distributed simulation of individual behavioral models. Nonetheless, such accelerator-based parallelization is constrained by the size of the physical memory of the accelerating hardware platform, *e.g.*, the GPU card.

2.7 Conclusion

In this chapter, we present a complete application-level study of using GPUs to accelerate data-intensive document clustering algorithms.

We first propose a hardware-accelerated variant of the TF-IDF rank search algorithm exploiting GPU devices through NVIDIA's CUDA. We then develop two highly parallelized methods to build hash tables, one with and one without support of atomic instructions. Even though floating-point calculations are not dominating this text mining domain and its text processing characteristics limit the effectiveness of GPUs due to non-synchronized branching and diverging, data-dependent loop bounds, we achieve a significant speedup over the baseline algorithm on a general-purpose CPU. More specifically, we achieve up to a 30-fold speedup over CPU-based algorithms for selected phases of the problem solution on GPUs with overall wall-clock speedups ranging from six-fold to eight-fold depending on algorithmic parameters.

We further extend our work to a broader scope by implementing large-scale document clustering on GPU clusters. Our experiments show that GPU clusters outperform CPU clusters by a factor of 30X to 50X, reducing the execution time of massive document clustering from half a day to around ten minutes. Our results show that performance gains stem from three factors: (1) acceleration through GPU calculations, (2) parallelization over multiple nodes with GPUs in a cluster and (3) a well thought-out data-centric design that promotes data parallelism. Such speedups combined with the scalability potential and accelerator-based parallelization are unique in the domain of document-based data mining, to the best of our knowledge.

Chapter 3

GStream: A General-Purpose Data Streaming Framework on GPU Clusters

3.1 Introduction

Stream processing has established itself as an important application area that is driving the consumer side of computing today. While traditionally used in video encoding/decoding scenarios, other application areas, such as data analysis and computationally intensive tasks are also discovering the benefits of the streaming paradigm. High computational demands by streaming have been met by general-purpose architectures via multicores. But with no end in sight for these inflating demands, conventional architectures are struggling to keep up. We already see significant increases in power and resource management costs particularly for homogeneous general-purpose multicores. Heterogeneous architectures with accelerators, such as GPUs, offer a viable alternative to meet the demand in computing as they deliver not only higher cost and power efficiency but also higher performance and scalability.

These performance potentials of GPUs originate from architectural design and programming strategies in favor of massive data parallelism. Today's latest generation of GPUs features hundreds of stream processing units capable of supporting much more data parallelism than a CPU does. The NVIDIA GPU programming model CUDA encourages users to create light-weight software threads at the scale of tens of thousands, which is orders of magnitude larger than the maximal hardware concurrency inside the GPU. This over-subscription of software threads relative to the hardware parallelism allows latency hiding mechanisms to be realized that mitigate the effects of the memory wall [120].

Existing streaming models, such as Lucid [118], LUSTRE [24] and SIGNAL [54] (see [109] for a survey), strive to provide a comprehensive streaming abstraction on one end of the spectrum. They are designed to be architecture-independent and focus on generality. This comes at a price as efficient execution under different computing platforms becomes an afterthought. On the other end, various compiler techniques and runtime systems were developed to map streaming abstractions to specific

hardware, *e.g.*, StreamIt [89], Brook [23], Cg [83] and Auto-pipe [30].

In this work, we consider a language extension and run-time system approach to map streaming abstractions to GPU *clusters*. Efficient utilization of resources in a GPU cluster is an essential prerequisite for its adoption in streaming domain, especially for large scale, data-intense applications. However, programming the state-of-the-art GPUs is not as flexible as programming CPU clusters. More specifically, we experience two challenges to achieve both programmability and performance:

(1) Deep (multi-level) memory hierarchies in a typical loosely-coupled GPU cluster connected via network interface pose a challenge. It takes multiple hops to transfer data in one GPU to another: first between the GPU device and host memory, then over distributed memory spaces onto a different node. The obligation to manage memory and data transfers exerts a burden to programmers, especially in a system where data flows are complicated. Recent work to mitigate this problem ([76, 110]) still falls short in that it exposes programmers to the underlying communication topology.

(2) Performance objectives between GPU parallelization and stream specifications tend to conflict with one another. On one hand, the GPU architecture is optimized for throughput making it applicable for latency-tolerant applications. On the other hand, many stream systems consider response time as the key performance metric. A delicate trade-off is necessary to incorporate GPUs as accelerators for such stream systems in order to meet requirements imposed by this metric.

Our GStream framework provides language and run-time support as a first-order design objective to map streaming abstractions onto GPU clusters. It addresses the aforementioned challenges in two complementary ways:

(1) GStream provides a unified memory transfer interface in the context of streaming data flows. No matter where source and destination of streaming data reside, the run-time system automatically performs the necessary memory copies or initiates message passing to guarantee data coherence. As a result, users no longer need to write explicit MPI messaging or CUDA memory copy directives. This feature greatly reduces development time.

(2) GStream provides an elastic data API (stream push/pop) to dynamically adapt the batch size for each GPU kernel. This is based on the observation that many streaming steps allow re-sizable input size. By applying this technique at runtime we are able to handle streaming systems that have dynamically fluctuating data-flow characteristics without sacrificing response time requirements.

Our GStream framework can be easily integrated with third-party CUDA libraries. This is motivated by the trend to provide GPU kernels that can accelerate hot-spots even with complex dependencies and synchronizations. Past GPUs thrived on naive data parallelism without synchronization between fine-grained data operations. Compilers can exploit such embarrassingly parallel algorithms by detecting idem-potent operations where output is a strict function of prior input flow without referencing any immediately preceding output. In modern GPUs, synchronization is natively provided at a fine-grained level. Consequently, numerous algorithms, including but not limit to linear algebra functions [1], FFT [2] and physical dynamics [35], can be significantly accelerated by CUDA abstractions and libraries.

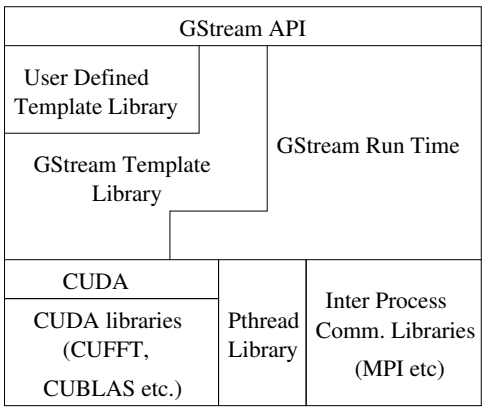


Figure 3.1: GStream Software Stack

Nonetheless, today's most efficient CUDA implementations are still hand-written codes. Being able to reuse these capabilities is considered crucial both from the performance and productivity points of view.

Overall, GStream is a general-purpose, scalable run-time framework that allows application to be expressed as streaming problems and efficiently executed on GPU architectures. In GStream, single program, multiple data (SPMD) codes are executed on a cluster of accelerated machines. Here, GStream provides transparent streaming data transmissions and automatic memory synchronization while offering users full control to utilize computational resources of both CPUs and GPUs.

An overview of the GStream software stack is shown in Fig 3.1. GStream combines software abstractions with concrete implementations targeted at different levels of parallelism: CUDA and CUDA-derived libraries for data-parallelism; POSIX thread abstraction for task parallelism in shared-memory; and inter-processing communication libraries for data sharing across distributed-memory machines. The later two components are completely concealed by the GStream run-time system. They can be replaced by any other libraries that provide similar functionality without affecting the application code base. For instance, we utilize the message-passing functionality of MPI for inter-node communication. Similar implementations can be built on top of other inter-node communication libraries (e.g., TCP sockets). GStream's run-time system integrates library components and completely hides the thread management and data movement from the user.

The contributions of this paper are the following: (1) We propose a novel streaming abstraction dedicated for GPU clusters. (2) The streaming data-flow abstraction is made extremely concise, intuitive and can be supported by existing language abstractions (instead of inventing yet another language). It hides from user the complexity of memory transfers between different address spaces. (3) The validity of the abstraction reaches well beyond streaming as illustrated via sample implementations for various domains, including data streaming, data parallel problems and numerical codes.

The rest of the chapter is organized as follows. The system model and design goals are stated in

Section 3.2. In Section 3.3, we describe the GStream API and its usage in detail. We present our system design in Section 3.4, experimental results in Section 3.5, related work in Section 3.6 and summarize the work in Section 3.7.

3.2 Design Goals and System Model

3.2.1 Design Goals

The focus of this work is to provide a general-purpose streaming framework dedicated to GPU architectures. We aim at satisfying several design goals:
(1) Scalability: The targeted platform is a cluster of machines accelerated by GPUs. There is no restriction on the size of the cluster.
(2) Transparency: Both the task scheduling and the GPU/host memory management for streaming data should be handled by the run-time system without any user intervention.
(3) Extendability: The library should be made extendable to meet customized needs while providing basic functionality.
(4) Programmability:The language syntax should be concise and type checking should be done at the compiler time.
(5) Flexibility: The computation cores can be chosen to freely execute on either CPU or GPU platforms. This allows fast prototyping with full debugging support on CPUs first.
(6) Re-usability: The cost of developing high-performance code on GPUs is higher than on general purpose microprocessors. Being able to reuse existing libraries will be a significant benefit.

3.2.2 System Model

Streaming systems are better understood when their internal data flow is characterized and analyzed. While efforts to specifically target certain architectures have led to different semantic abstractions of streaming, two fundamental components are common to typical streaming systems: data processing units and data links that connect them. In GStream, we refer to these as *filters* and *channels*, respectively.

Filters consume zero, one or multiple streams of data types and similarly produce any number of streams of identical or dissimilar data types. Filters without input or without output are referred as *source filters* or *sink filters*, respectively. In GStreams, source and sink filters are not differentiated from any other filters.

Channels exist whenever there is a data flow between filters, *e.g.*, one filter's input stream originates from another filter's output stream. From the filter's point of view, channels are effectively unbounded queues. Two types of channels are differentiated in GStream: point-to-point (p2p) channels and group channels. P2p channels are uni-directional and used for ad-hoc data transmissions. Each p2p channel has a predecessor filter and a successor filter. In contrast, group channels have well-defined group

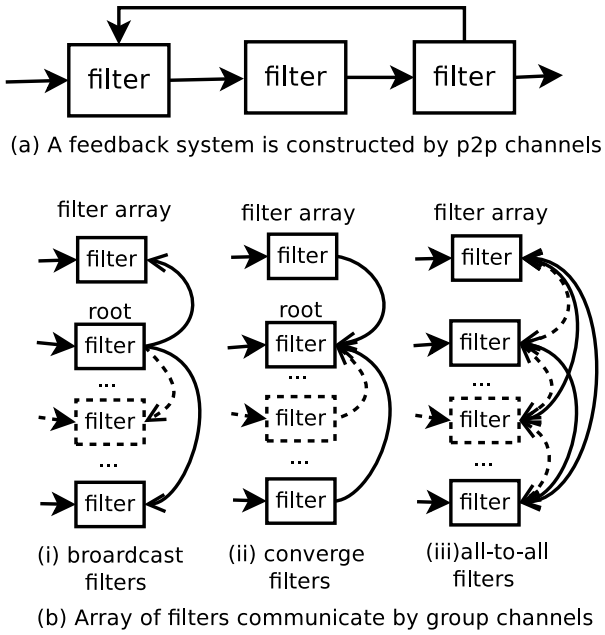


Figure 3.2: System Model

behavior and are used for inter-node data transmissions. GStream currently supports broadcast, reduce and all-to-all group channels. Both group and P2P of channels are strongly typed, connected to filters via *ports* and are associated with unique port IDs on filters. The operation of data on channels are realized via a simple push/pop interface (see Section 3.3.2).

With the definitions of filters and channels, we can build customized streaming applications in a multi-node environment. For example, we can construct a standard pipelined system consisting of just P2P channels (Fig. 3.2(a)). Furthermore, backward channels are supported to realize feedback systems. Another dimension is given by arrays of filters (Fig. 3.2(b)), where each array element resides on a different node. The communication between filter arrays can be facilitated by group channels but P2P channels are supported as well.

The hybrid model of channels allows users to combine flexibility with productivity. On one hand, the P2P channel abstraction makes it possible to build any kind of stream graph. On the other hand, group channels prevent the programmer from having to build well-defined and widely-used communication patterns from scratch, which is a tedious and error-prone task.

3.3 GStream Overview

GStream is a C++ template library for data parallel, data streaming applications based on the streaming abstraction described in the previous section. Using C++ has two major advantages: (a) It seamlessly

integrates with existing frameworks including CUDA and MPI; (b) The template meta-programming feature in C++ provides an ideal technique to make the library reusable and expandable. In the following, we will (1) present the key characteristics of filters and channels in GStream; (2) show the principle GStream API and (3) illustrate the steps to write a program (as streaming specifications) in GStream.

3.3.1 GStream Abstraction and Convention

GStream makes several assumptions to abstract a streaming system. The basic computation unit is a filter. Filters can run independently from each other once their input data is available on an input channel. The main body of a filter is generalized into a three-step pattern (see Fig. 3.3). The start() and

```
void Filter::run() {
    start();
    while (!isDone())
        kernel();
    finish();
}
```

Figure 3.3: Filter Specification Pattern

finish() functions are executed once at the beginning and the end of the filter life cycle. They are used to execute chores such as parameter initialization and internal resource allocation/deallocation. They can also be used to allocate/deallocate scratch space, which is encapsulated in the filter itself. The central activity of a filter body is the kernel() function. Inside the kernel() function, a filter typically executes as follows: It waits for tuples from input ports, processes data and generates output tuples to output ports. A sequence of these steps is called a batch process. Batches continue to execute until input data is exhausted (or run forever if inputs are infinite streams). One of the differences between GStream and other streaming abstractions, such as StreamIt [113], is how the parallelism is defined as filters. In StreamIt, the user needs to statically define the behavior of a filter on the most fine-grained unit. In contrast, the *filter parallelism* in GStream is defined as a range of tuples a filter can process in one batch. This can be decided at run-time to give users more flexibility to control data flows. This design caters to dynamic scenarios where the batch size can change at run-time. Such variance is controlled by the user through two APIs: getMinDegree(portId) and getMaxDegree(portId) define the legal range of the number of input tuples per port. The user is then required to provide a general routine that can successfully handle input tuples in this range. On the output side, the number of tuples to be generated is determined by the size of the input tuples a filter receives in a batch. Making fine-grained filter behavior transparent and flexible through massive parallelism is precisely what distinguishes GStream from other streaming abstractions.

P2P channels and group channels are exposed to the user at different levels. P2P channels are

explicitly constructed by the pipe operator | of the filter class. Group channels do not require any user intervention. They are associated with predefined special filter arrays. Their construction is handled internally and transparently by GStream. The advantages of using the pipe operator to express P2P channels are its conciseness and intuitive notation.

```
(1) f|g|h; // simple filter pipeline
(2) h|f;   // extend (1) with feedback
(3) for (i=0;i<M;i++) a[i]|b[i]; // filter array
(4) for (i=0;i<M;i+=2) {
      a[i]|c[i/2];a[i+1]|c[i/2]; } // merge
```

In example (1), a pipeline of filters can be expressed in just one line of code by interlacing filters and pipes. Specification of a feedback path requires (2) just one extra line of code. Arrays of filters (3) can also be linearly combined or by flow splits or merges (4).

3.3.2 GStream APIs

The list of filters is maintained internally by the GStream run-time. Different filters in GStream are defined via concrete classes derived from the same base class (see Fig. 3.5) with basically three predefined virtual functions: void start(), finish() and kernel(). It is not necessary to override the start() and finish() functions if their bodies are empty. Similarly, getMinDegree(portId) and getMaxDegree(portId) have default values (1 and 4096, respectively) that can be overridden by the user to specify a different range.

Streaming data is owned and managed by channels through a simple channel interface. We found that a simple data push and pop API suffices to express data processing. Pop extracts streaming data from input channels. Conversely, push APIs injects data on output channels. To stream data out of an input channel, pop() is first called to obtain the buffer pointer. The call is blocked if the channel does not contain the number of tuples in the range defined by the minDegree and maxDegree on this port (unless end-of-stream is reached where the stream is flushed unconditionally). Upon returning, the run-time system supplies the current maximal number of tuples that satisfies the given range. As soon as the user has consumed the input, pop_finalize() can be invoked to inform the runtime that the channel can safely release the input. Similar two-phase operations apply to the output channel. This two-phase API requires a strict pairing of API calls by the user, which results in a number of benefits:

- Unnecessary memory copy operations are avoided. For instance, in the push API, reserve() is first called to obtain the current memory pointer of a channel. Once the data is ready, reserve_finalize() is called to signal the availability of the data. In contrast, if only reserve_finalized() were provided, the user would need to allocate memory explicitly.

- The size of reserved/popped data during the first step is not necessarily the same as the finalized size in the second step (but always greater or equal). This addresses the case when peek size differs from pop size.

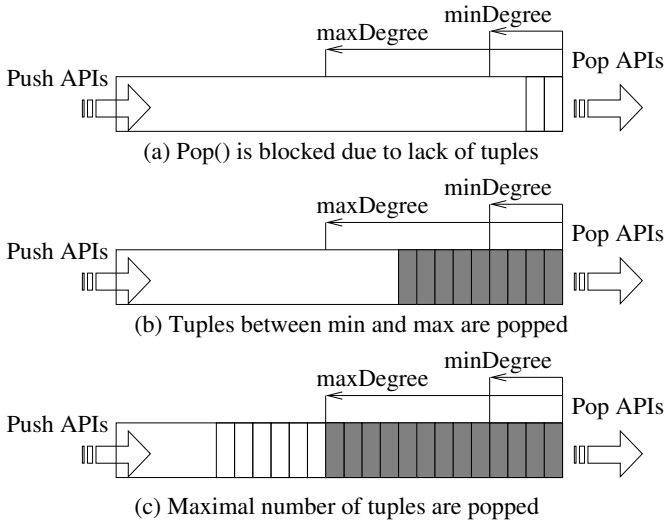


Figure 3.4: Schematic of Elastic Pop APIs

- Irregular data types or types with unknown prior size (*e.g.*, IP packets) can be handled by casting channel types to unit type characters. When the actual tuple size is unknown during the first step, the consumption size can be adjusted in the second step.

An illustration of the elastic pop API is given in Fig. 3.4. For each channel, the runtime system maintains a buffer viewed as an unbounded queue of tuples. When the downstream filter calls a pop() with an acceptable range, the runtime system checks the availability of data on the channel. If the number of tuples is less than the minimal range, the call is blocked (case (a)). If the number is greater than or equal to the minimal threshold, the call returns indicating the actual number of elements (capped by the maximal threshold). The system maintains the integrity of the popped data until the pop_finalize() is called. Only thereafter can the buffer be reused by the channel to store new pushed data.

3.3.3 Case Study – A Finite Impulse Response (FIR) Filter

An example of a simple FIR filter expressed in GStream is shown in Fig. 3.6. A FIR filter is a specific aggregate filter with a sliding window of order m (see Fig. 3.6(a)). We create a pipeline of three filters: a random number generator, the FIR filter and a print filter. In the main program (Fig. 3.6(b)), we demonstrate how the three filters are initialized and added to the stream system (lines 3 to 11). The P2P channel connection is expressed concisely as a single line (line 14). Both the random number generator and print filters are provided by the template library. The FIR filter definition is shown in Fig. 3.6(c).

Lines 5 to 7 override the start() function to set up the coefficient array. Method getMinDegree() needs to be overridden (lines 8 to 10) because it takes at least m input tuples to generate the first output

StreamSystem API:
void addFilter(FilterBase *filter); void run();
Major Filter Functions:
void kernel()*; void start()+; (empty by default) void finish()+; (empty by default) int getMinDegree(int portId)+; (return 1 by default) int getMaxDegree(int portId)+; (return 4096 by default) void assignToNode(int nodeId); /* for multi-node case */ void setToUseGpu(); /* set to use GPU or CPU */ *: must override. + has default behaviors
Channel Push API:
void reserve(StreamChannelBuffer &buffer, int size); void reserve_finalize(int size);
Channel Pop API:

int pop(StreamChannelBuffer &buffer, int min, int max);
void pop_finalize(int size);
void waitForAny();

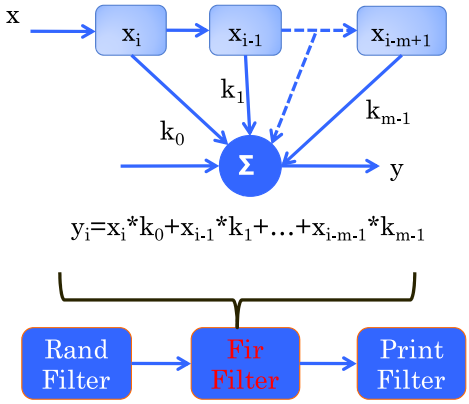
Figure 3.5: GStream API

tuple, where m is the degree of the FIR filter. Line 11 to 31 depict the execution of the main body of the FIR kernel function. It keeps popping data from its input port (lines 16 to 17). The returned size (int batch in line 16) always falls in the provided range of [getMinDegreel(0) ... getMaxDegree(0)]. The GStream runtime system guarantees continuous storage for the data in memory. Once the input size is known, we can use the information to reserve a buffer on the output channel (line 19). After the computation (lines 20 to 22) is completed, the output is pushed to the output port (line 26). To add GPU support, all we need to do is to replace the CPU code from lines 20 to 22 with a GPU kernel call. Any other code sections remain unchanged.

3.4 Design and Implementation

We have implemented the GStream library using C++ programming language features with extensive use of template-based generic programming techniques [9]. GStream is deployed on a cluster of nodes, each equipped with a GPU.

With a template tool for manipulating collections of types (a typelist template of the Loki library [4]), we design the filter class to realize the filter abstraction in GStream. The base filter class (*Filter<inputTypeList, outputTypeList>*) is an abstract template class. It contains two templates as the filter's input type list and output type list. Filters are mapped to different threads and executed independently from one another.



(a) Fir Filter and GStream Structure

```
int main()                                          1
{                                                   2
  StreamSystem ss;                                  3
  RandFilter<float> rf;                             4
  FirFilter<float, 100> firf;                       5
  PrintFilter<float> pf;                            6
                                                    7
  /* add filters to system */                       8
  ss.addFilter(&rf);                                9
  ss.addFilter(&firf);                              10
  ss.addFilter(&pf);                                11
                                                    12
  /* construct p2p channels */                      13
  rf | firf | pf;                                   14
                                                    15
  /* ready to run */                                16
  ss.run();                                         17
  return 0;                                         18
}                                                   19
```

(b) Main Program

```
1  template<typename T, int m>
2  class FirFilter:public Filter<typelist1<T>, typelist1<T
       >>
3  {
4   public:
5    virtual void start() {
6      ... /* setup coefficient array k[m] */
7    }
8    virtual int getMinDegree(int) {
9      return m; // overwrite the default return value 1
10   }
11   virtual void kernel()
12   {
13     StreamChannelBuffer<T> input;
14     StreamChannelBuffer<T> output;
15     /* pop inputs of size from m to getMaxParallel(0) */
16     int batch = inputPort[0]→pop(&input, getMinDegree
           (0), getMaxDegree(0));
17     if (batch != −1){
18       /* reserve output buffer */
19       outputPort[0]→reserve(&output, batch − m + 1);
20       for (int i = 0; i != m; i++) {
21         ... /* the kernel calculation, omitted */
22       }
23       /* output data ready, finalize the reservation */
24       outputPort[0]→reserve_finalize();
25       /* only pop (batch − m + 1) from the input port */
26       inputPort[0]→pop_finalize(batch − m + 1);
27     } else { // returning −1 indicates the end of stream
28       setDone();
29   } }
30   private:
31    T k[m];
32 };
```

(c) Fir Filter Class Definition

Figure 3.6: Fir Filter Example

The template design ensures the objectives of high programmability and extendability (see Section 3.2.1). It provides enough flexibility for users to customize a filter's behavior. The derived filter is required to define its own kernel() function as a pure virtual function. The void start() and finish() functions can be optionally overridden if internal state needs to be initialized or resources need to be allocated/deallocated. The library currently contains several pre-defined filters such as a random number generator, a printing filter and a hash/map filter. Users may add new filters by defining new classes derived from the base filter class.

The design of a kernel() function gives the user wide flexibility, but it usually adheres to the following pattern:

```
out_channel->pop();
in_channel->reserve();
kernel_calculation();
out_channel->pop_finalize();
in_channel->reserve_finalize();
```

The kernel_calculation() is the core computation that can be implemented by mapping the kernel to either a GPU (via CUDA) or a CPU. It can also be replaced by library calls, including numerical GPU libraries such as CULA, which meets the reusability objective of Section 3.2.1.

Every port in a filter is associated with a data type, and the data type needs to be incorporated in the filter class' typelist template. This ensures strong type-checking at compile time. But the limitation is that filters cannot have an arbitrarily large number of fan-in/out ports. We address this limitation by supports for (a) grouping of channels (group channels take only one port id, even if there are multiple data links); (b) creation of intermediate filters in a tree structure; and (c) combination of data types of multiple ports into one complex data type.

To meet our objective of flexibility (see Section 3.2.1), a method setToUseGpu() in filter is provided to indicate that the computation routine should be accelerated by a GPU. By default, streaming data of such a filter resides within the GPU address space. This call acts as a hint to GStream to automatically perform necessary DMAs. Filters are assigned to a particular node by calling the assignToNode() member function. If two concatenated filters are assigned to different nodes, a pair of asynchronous MPI send/recv calls are setup to realize the channel pop/push interface. Each channel is associated with a data type, which matches one of the types in the filter class' input/output typelist according to the channel's port id in the filter. Internally, a channel has two buffers, one each for CPU and GPU. Depending on the receiver's filters property, the run-time system automatically synchronizes the memory.

The overall system design for a GPU cluster is illustrated in Fig 4.3. The resulting generated executable is an SPMD program. All filters assigned locally are instantiated by a CPU thread on a node. The GPU is time-shared among all filter threads: a global command FIFO queue is maintained for the GPU. All GPU-related operations issued by filters, including GPU memory allocation, DMA memory

copy and kernel executions, are pushed to the queue. We thereby realize the transparency objective (see Section 3.2.1) as scheduling, memory management and memory movement are automated.

A dedicated GPU thread serves the FIFO queue when the queue is non-empty. For nodes awaiting stream data from a different node, an upstream thread is created to listen to the network messages from other nodes. Once data is being received, the thread will push the data to the corresponding local filters. All MPI calls are asynchronous to avoid the deadlocks (e.g., due to blocking MPI send/receive orderings triggered by filter dependencies). GStream currently does not manipulate the execution order of the GPU FIFO queue. The mapping of filters to physical nodes is performed manually by the user through the filter::assignToNode(int nodeId) API. Since the underlying data transfer is made completely transparent to the user, users can experiment with different layouts via rapid prototyping to determine the best configuration. One of our future work is to automate the process by assigning nodes with high bandwidth streams to the same node.

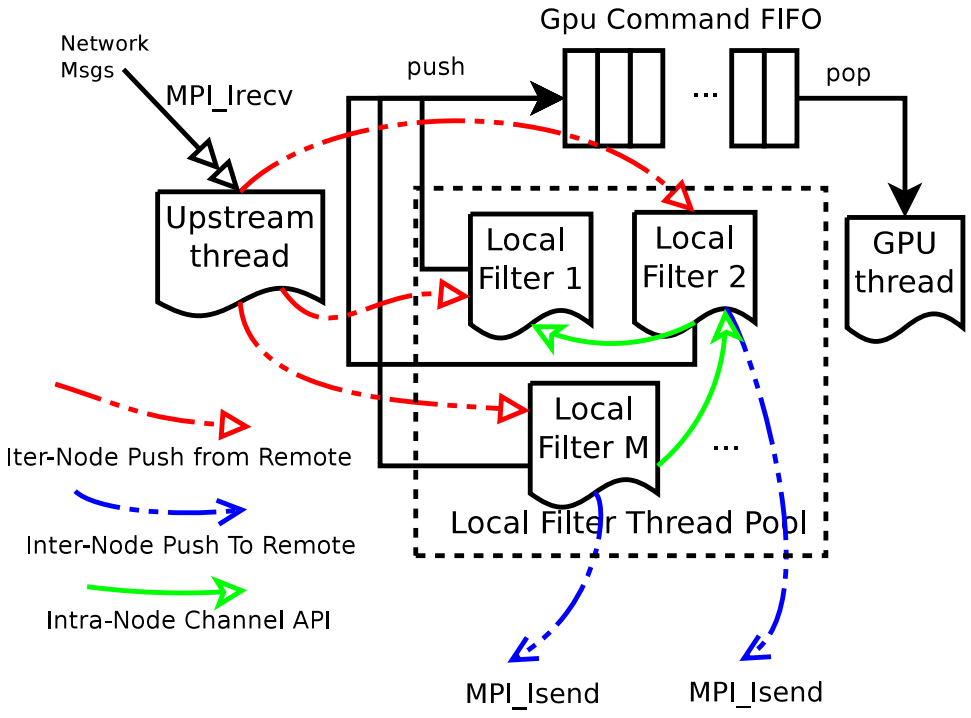


Figure 3.7: System Overview

3.5 Experimental Results

We performed experiments on a cluster where we utilized up to 32 nodes equipped with GPUs and connected by QDR Infiniband (36 Gbps). Each node consists of two AMD Opteron 6128 sockets (16 cores per node) and an NVIDIA Tesla C2050 graphic card. GStream is compiled with the CUDA 3.2 compiler combined with OpenMPI for MPI-style communication method.

3.5.1 Streaming Micro Benchmarks

We have implemented several representative streaming and non-streaming, iterative benchmarks using GStream, namely FIR filter, matrix multiply (MM) and FFT. These benchmarks require no more than a few filters, including a pre-defined random floating-point generator filter and a terminal output filter. For each benchmark, we provide four implementations: (a) A native C/C++ program running on CPUs without considering any streaming behavior (but still uses third-party libraries); (b) a multi-threaded C/C++ program using the GStream library *without* GPU support; (c) a native CUDA implementation without considering streaming behavior and (d) GStream *with* GPU support.

The filter construction of the three benchmarks is shown in Fig. 3.8(a)(b)(c). GStream makes it straightforward to run filter arrays on multiple nodes to increase the throughput. We were able to run 32 copies of the filters on 32 nodes. The speedups of all implementations running on 32 nodes are shown in Fig. 3.9, with implementation (a) chosen as the baseline. The performance ratio of (b) over (a) indicates the overhead of the GStream run-time system, which is negligible as the ratios for all benchmarks are very close to one. In the following, we discuss each micro-benchmark in detail, including the detailed application parameters and third-party libraries we have used.

FIR has been introduced as a code example previously. We set the order of the FIR filter to 100, indicating an aggregate filter with a sliding window of size 100. A hand-coded FIR GPU kernel is developed in this benchmark. GStream using a GPU achieves a speedup of about a factor of 6 over the vanilla C version on a CPU.

For the matrix multiply (MM) benchmark, we measure the time to calculate a sequence of multiplies on square matrices of dimension 1024 by 1024. Both (c) and (d) integrates the CUBLAS library [1], an efficient implementation of BLAS on top of CUDA. For accessing the vanilla C Version on a CPU, the Template Numerical Toolkit (TNT) [5] is used for comparison.

Similarly for FFT, both the original C program and the CPU version of the GStream implementation use FFTW, a widely used and highly efficient FFT library. CUFFT [2], a FFT CUDA library shipped along with the CUDA SDK, is used in GPU evaluations. In this test case, the performance of a 2D (512 by 512) single-point complex FFT is compared.

Of these three benchmarks, the GPU version of GStream offers 3 to 27 times speedup over the corresponding C version. The CPU version of GStream outperforms the C program for FIR and FFT in spite of the synchronization overhead. This is because filters in GStream are executed in multiple

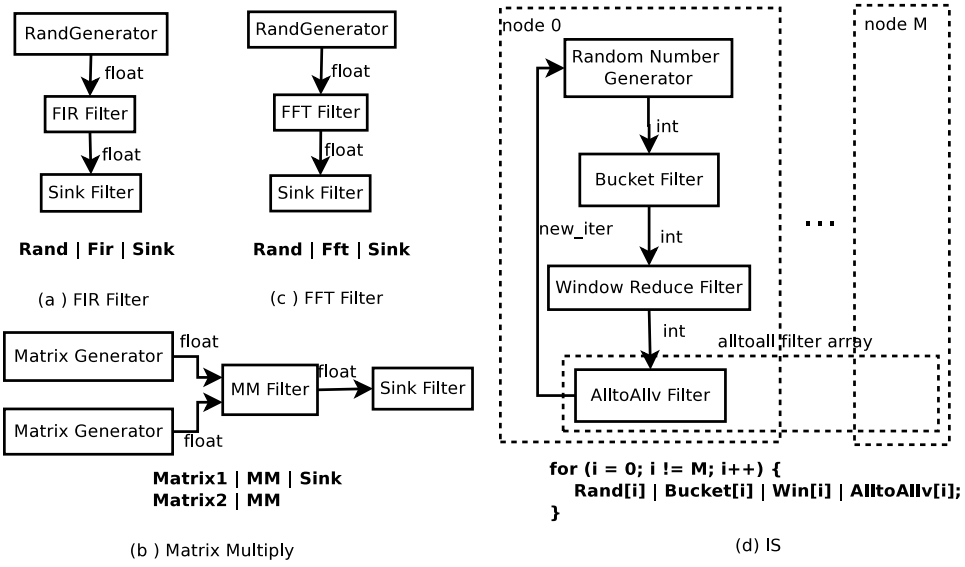


Figure 3.8: Filter Structure for Benchmarks

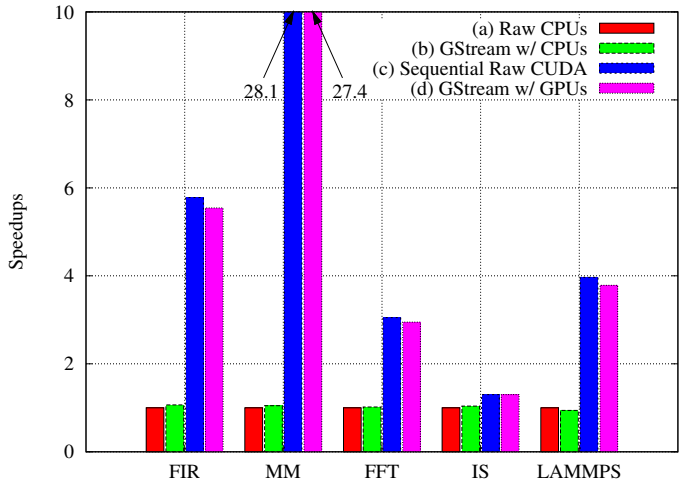


Figure 3.9: Speedup of Benchmarks on 32 CPU/GPU Nodes

threads. The random number generator filters in these two benchmarks execution is overlapped with FIR filter. This parallelism can compensate for the overhead of the library. The ratios of (b) over (a) and (d) over (c) show that GStream imposes little overhead to the overall system.

3.5.2 Scientific Benchmarks

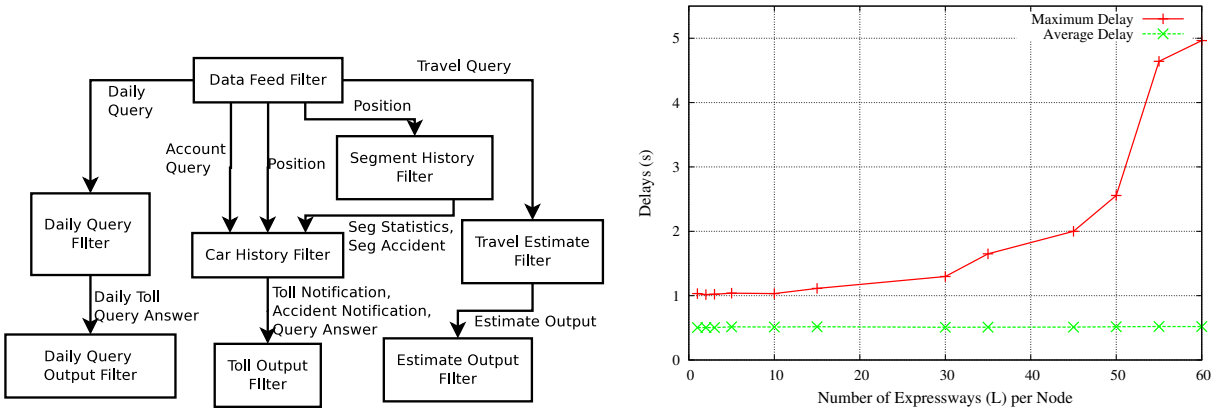
We rewrote the IS (integer sort) benchmark of the NAS parallel benchmarks [17] and converted it into a filter-based program. The filter structure is depicted in Fig. 3.8(d). Input integer numbers are produced by the Random Number Generator. The Bucket Filter consumes these integers in large batches and calculates the bucket statistics of each batch. The Window Reduce Filter summarizes bucket information over batches until all inputs are processed. The final bucket statistics is fed to an Alltoallv Filter for post processing. Both Bucket and Window Reduce Filters can be mapped onto GPUs or CPUs. The GPU version is slightly faster than the original benchmark (see Fig. 3.9 for class D on $M = 32$ nodes). This is because IS is a communication-bounded benchmark, which limits GPU benefits.

GStream can be integrated with legacy codes by exposing APIs such as addFilter() and addChannel(). We have integrated GStream into LAMMPS, a molecular dynamics simulator distributed by Sandia National Laboratories [97]. LAMMPS is designed as a computing platform for simulating soft materials, solid-state materials and coarse-grained or mesoscopic systems. The original code runs on single processors or in parallel systems using MPI. More recently, accelerators, such as GPUs, have been deployed and are supported by LAMMPS as an effort to reduce the total computation time [35]. The simulation in LAMMPS is organized as a pipeline of computational steps making it a perfect candidate to apply our GStream concept.

In this case study, we replaced the LJ (Lennard-Jones) potential cutoff step with a customized filter in GStream and added channel manipulations in the LAMMPS source code to trigger its execution. The last set of bars in Fig. 3.9 shows the speedup of using the original GPU code and the GStream implementation vs. the CPU implementation on 32 nodes. Again, the overhead of adding the GStream library is negligible. We have only converted one hot-spot of the entire pipeline into GStream filters at this time. Complete transformation of all pipeline steps to GStream would result in a code base that is better organized and more expandable. In general, the ease of integration within legacy codes step-by-step for each kernel, such as demonstrated with LAMMPS, provides a graceful transition that facilitates the adoption of the GStream in other domains, such as complicated numerical codes.

3.5.3 Linear Road Benchmark

A widely-used real-time streaming benchmark is the Linear Road Benchmark [13], originally proposed to provide a scalable and fair benchmark for Stream Data Management Systems (SDMS). It simulates a toll system of motor vehicle expressways of a large metropolitan area. Expressways are divided into one-mile-long segments. The system needs to keep track of the number of vehicles, detect accidents



(a) Linear Road Benchmark Filter Structure per Expressway

(b) Output Delays per Expressway

Figure 3.10: Linear Road Benchmark on 32 CPU/GPU Nodes

in each segment and determine toll charges for each vehicle. In the meantime, queries such as vehicle balance, historical charges and travel time estimation need to be answered. In a three-hour simulation, the response time of each output is measured with regard to the following timing constraint: Outputs need to be produced within 30 seconds for travel time estimation queries and within 5 seconds for all other events, especially for toll notifications, which are on the critical path of the system. The performance metric of an implementation is then given as the maximal number of expressways, L, without violating the timing constraint.

We have implemented the toll query system using the streaming abstraction of GStream, which allows us to focus specifically on customized filter design in the system. The filter graph per expressway is illustrated in Fig. 3.10(a). A data filter feeds the input tuples according to timestamps to mimic a real-world scenario. Inputs are filtered here to direct streams to different filters. Position reports are transferred to a segment history filter to generate segment statistics and detect potential accidents. The same position reports are also fed to a car history filter to determine if the car has entered a new segment. A channel connects the segment history filter with the car history filter. This channel is activated to transfer segment statistics (number of vehicles in the last minutes, accident flags) every minute to assist the car history filter in determining toll charges. Vehicle accounts are kept in the car history filter. Therefore, account queries pass through the car history filter, too. Other queries (daily queries and travel queries) are processed independently via a separate data flow.

Once the filters and their data dependencies are finalized, we can freely experiment with different filter mappings into our physical node space due to the transparency of data transmission provided by the GStream run-time system. The highest performance was obtained by assigning filters belonging to one expressway to the same physical node. “L-rating” defined as the maximal number of expressways supported by a system meeting response time constraints delimited by 5 seconds. The response time in

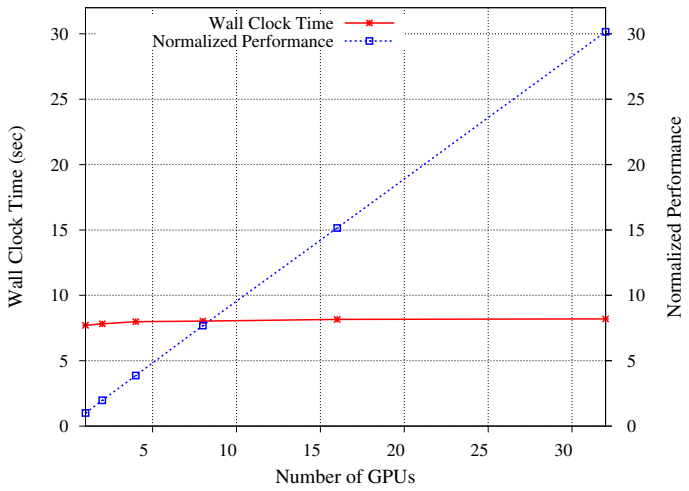


Figure 3.11: Weak Scaling in 3D Stencil on up to 32 GPUs

one node at different number of expressways is shown in Fig. 3.10(b). In total, we get an L-rating of $60 \times 32 = 1920$ on 32 GPUs. As a comparison to previous work, both Aurora [8] (2003) and SPC [71] (2006) achieved L-ratings of 2.5 on a single machine. The most recent implementation in SCSQ [123] (2010) reported an L-rating of 64 in a dual quad-core Linux cluster.

3.5.4 3D Stencil

GStream can improve the productivity to write programs that are not considered as traditional streaming applications. In this experiment, we implemented a 5-point 3D stencil Jacobi iteration on 32 GPU nodes to demonstrate that our scalability objective is being met (see Section 3.2.1). Filters divide the stencil space along the Z axis. In each iteration, a filter needs to exchange its borders with two neighbor filters. We set each filter's stencil space to $512 \times 512 \times 512$. Fig. 3.11 depicts that the wall-clock time remains constant under weak scaling (with proportional increase of both the number of nodes and the problem size). As a result, the performance normalized to a single GPU increases linearly. The reason GStream experiences perfect weak scaling is due to the fact that inter-node channels are implemented by asynchronous MPI calls that can overlap with the internal computations. Furthermore, programmers only need to focus on the development of stencil kernels on GPUs since communication is transparently handled by GStream.

3.6 Related Work

Stream processing has been studies for a number of decades [109]. In the earlier years, the data flow semantic models and languages to support them were the primary focus. Several Data Stream Management Systems (DSMS), such as TelegraphCQ[31], Aurora [8], Medusa [37] and the STREAM project [12] [89], focused on continuous query processing, which is only one example of GStream's more general applicability and expressiveness.

Our concept of filters is loosely inspired by StreamIt [113], a platform-independent streaming language and compiler environment. Our runtime-centric dataflow approach is more general than their static analysis and transformation methodology. In fact, GStream could be used as part of the runtime system to extend StreamIt to GPU clusters. We further embrace a coarser-grained data parallelism than StreamIt, which results in performance beyond prior work [55]. A number of other filter-based frameworks have been designed [90, 18, 111, 112, 128]. Similarly, they encapsulate computations into filters, a central concept to express algorithms. But their designs are based on different objectives to fit a specific domain that they target. They also tend to target shared memory while we consider filters in a distributed memory environment across compute nodes in a cluster.

Brook [23] is a streaming language dedicated to GPUs. It does not support scheduling across kernels. It relies on a sequential language to trigger a streaming process. Udupa *et al.* [114] extended the ideas of StreamIt with a direct port to a single-node GPU platform. Filters are mapped to a sub-kernel level abstraction to realize transparent scheduling. GStream takes streaming to another level by combined support for *coarse-grained* data parallelism and filter arrays to target *multiple* GPUs. CUDA supports simple stream objects for command sequences that execute in order. While this concept matches simplistic pipelined computations, it fails to generalize to non-pipelined execution patterns and lacks support for expressing more complicated data dependencies that are widespread.

Recent years have witnessed many efforts to provide unified programming models or language support for accelerators including GPUs. StarSs [15] takes a pragma-based approach to express computational kernels as *tasks*. StarPU [14] uses *codelets* as an abstraction of a task that can be mapped to an accelerator. Both of them offer a certain degree of scalability but they are strictly constrained to the shared-memory paradigm. A new language called the X code is proposed in [30]. It contains a set of automated tools (Auto-Pipe) to aid in the design, evaluation and implementation of applications that can be executed on acyclic computational pipelines. It shares with GStream the philosophy that data flow should be expressed at a higher level to remove user interference. However, its scalability in larger clusters has not been shown, to the best of our knowledge.

3.7 Conclusion

We have designed and implemented GStream, a general-purpose, scalable data streaming framework designed for clusters of GPUs. GStream is inspired by a lack of streaming abstraction dedicated to

massively parallel architectures and their suitability to express data parallelism. We presented a novel and concise, yet powerful streaming abstraction amenable to GPUs. Communication patterns are expressed as point-to-point channels or as group channels. This abstraction ensures flexibility in runtime adaptation and fosters productivity during coding by letting programmers focus on the description of data organization and operations performed on the data without explicitly expressing task parallelism constraints. Programmability is realized through extensive use of template-based generic programming techniques in C++, which fosters portability and integration with an existing code base.

Overall, GStream's strength is in its ease of use and its applicability to a variety of domains not constrained to traditional streaming problems, as demonstrated by our experimental results. These aspects combined with efficient exploitation of GPU resources have the potential for a GStream-like paradigm to succeed.

Chapter 4

Auto-Generation and Auto-Tuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters

4.1 Introduction

Main-stream microprocessor design no longer delivers performance boosts by increasing the processor clock frequency due to power and thermal constraints. Nonetheless, advances in semiconductor fabrication still allow the transistor density to increase at the rate of Moore's law. This has resulted in the proliferation of many-core parallel architectures and accelerators, among which GPUs quickly established themselves as suitable for applications that exploit fine-grained data-parallelism.

Still, software development for parallel architectures turns out to be more difficult than that for uniprocessors in terms of obtaining high performance, even when aided by new programming models such as CUDA [7] and OpenCL [69]. Programmers spend substantial time and effort to understand the underlying architecture to best utilize all resources. This can become a daunting task since performance is affected by a multitude of architectural features. Even worse, architectural difference between generations of the same hardware line may require a diversity of optimization strategies with sometimes opposite optimal set-points. Programmers may have to explore many (if not all) combinations of optimization options and parameter values to determine the best configuration for a particular hardware. This poses a great challenge since programmer productivity is adversely affected by lengthy tuning efforts. Simply re-profiling and re-writing the program each time the hardware is upgraded is neither desirable nor feasible over time.

Current compilers for general-purpose languages struggle to balance portability, performance and programmability. Domain-specific languages (DSLs), in contrast, offer a promising solution at the expense of sacrificing language generality [25]. DSLs have restricted expressiveness aimed at a particular

domain. It is precisely this domain-specific knowledge that allows the DSL-compiler to attain performance achieve comparable to hand-coded domain implementations. In contrast, general-purpose languages are inherently limited in their optimization scope in exchange for assuring correctness and good overall (but not best) performance on average for a wide range of applications. Examples of well-known DSLs are HTML for web pages, Matlab for scientific computation and SQL for database queries.

This work focuses on providing a portable source-to-source auto-generation and auto-tuning framework for iterative 3D Jacobi stencil computations on different GPUs. We generate stencil code as native CUDA code for NVIDIA GPUs, yet the same principles apply for GPUs of other vendor and comparable programming models, e.g., OpenCL [69].

Stencil (nearest-neighbor) computations are widely used in scientific computing, including structured grids as well as implicit and explicit partial differential equation (PDE) solvers in domain ranging from thermo/fluid dynamics over climate modeling to electromagnetics among others. An iterative explicit stencil computation is comprised of computation-intensive kernel. At each discrete timestep, all stencil points are updated according to values of their spatial neighbors from a previous timestep. On one hand, the uniform and communication-free behavior is well suited for the SIMT (single instruction multiple threads) paradigm advocated by state-of-the-art GPUs. On the other hand, an efficient GPU implementation is sensitive to neighbors accessing patterns across different stencils. One key characteristic of most stencil computations is the overlap in input values to update multiple neighboring points. Exploiting this property is crucial to achieve competitive performance on GPUs. One common GPU technique is to use the on-chip *shared memory* (shared by a warp/block of threads) as an intermediate storage space for overlapped input values. Instead of letting each thread fetching all inputs from off-chip global memory, all inputs are first cooperatively loaded to shared memory before they are referenced when computing a new stencil value. This is beneficial even in more recent generations of cache-enabled GPUs since this shared memory is orders of magnitude faster than global memory. It is curial to determine is how many threads should be grouped together in one block: Increasing the block size increases shared memory data reuse but may also deteriorate the GPU's occupancy rate of processing units [7].

There are many other factors that affect the performance. For example, how many stencil points should a thread work on? The larger the number, the more instruction-level optimizations can be applied by a compiler. But the less data-parallelism is exposed, the higher risk is for not fully utilizing a GPU's processing units. Also, is mapping inputs to texture memory faster? Our experiments show that the answer varies from case to case. Overall, there is no universal, optimal configuration for all types of stencil computations on different GPU models. Therefore, auto-tuning is not only desirable but also necessary to improve performance in this particular domain.

This heterogeneity across different generations/models of GPU exerts more challenges to programmers working on a cluster with hybrid models of GPUs. Such clusters becomes increasingly common due to the variety of GPUs on the shelf and incremental hardware upgrades. We further study the strategies to make the best use of all available GPU resources on homogeneous/heterogeneous GPU clusters

with optionally dissimilar parameters per distinct GPU type.

This work falls into the area of implicitly parallel programming models [70]. Our model relies on a compiler to generate highly efficient parallel code without requiring much interaction with the programmer.

The contributions of this work are:

- We abstract a wide variety of stencil computations into a set of domain-specific specifications. This allows the end-user to customize specific problems without having to consider the underlying architecture.
- We thoroughly summarize optimization techniques for stencil problems in previous literature and extract three sets of key parameters that affect the performance: (1) Block sizes that determine the shared-memory usage per block; (2) block dimensions that affect the number of registers consumed by each thread and (3) whether or not to map a subset of the input into texture memory.
- We develop an auto-generation and auto-tuning framework, i.e., we translate stencil specifications into executable code that is subsequently auto-tuned to the optimal configuration within a parametrized search space for each target GPU.
- We apply auto-generation and auto-tuning as a means for parameter optimization to GPU clusters and generate MPI program with identical parameters per GPU in homogeneous GPU cluster and with potentially dissimilar parameters per distinct GPU for heterogeneous GPU clusters.
- We show that heterogeneous GPU clusters exhibit the when leveraging *proportional partitioning* of the data space relative to single-GPU performance.
- Experimental results show competitive performance to manual tuning and demonstrate the superiority and necessity for auto-tuning to combining performance with correctness.

The rest of the chapter is organized as follows. The related work is presented in Section 4.2. In Section 4.3, we describe the stencil specification and the output of the framework. We explain various optimization strategies and how they are applied to our framework in Section 4.4. Detailed experimental results are presented in Section 4.5, with thorough comparison with previous works. We summarize our work in Section 4.6.

4.2 Related Work

Auto-tuning has long been identified as an effective approach to offer portability and productivity. For example, ATLAS [11], OSKI [117] and FFTW [52] are well recognized auto-tuning libraries targeted at general-purpose processors for dense/sparse linear algebra subroutines and FFT kernels in digital signal processing, respectively.

```
typedef struct {
  int dims[3];
  int iter;
  int haloMargins[2][3];
  ...
  int numNodes; // for multi-node
  int curNode;// for multi-node
} StencilConfig;
```

(a) Auto-Generated Code

```
initStencil(StencilConfig *config);
stencilIteration(StencilConfig *);
stencilIteration_mpi(StencilConfig *);
exitStencil(StencilConfig *);
```

(b) API

```
int main(int argc, char **argv) {
  StencilConfig config;
  config.iter = 0;
  config.dims[0] = 256; ... // more init.
  initStencil(&config);
  while(config.iter < 100) // run 100 iterations
    stencilIteration(&config);
  exitStencil(&config);
}
```

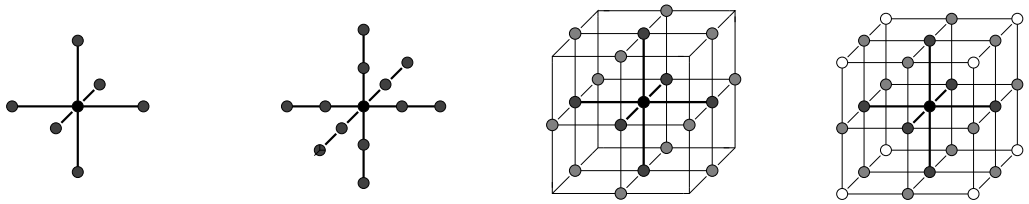
(c) Sample User Code

Figure 4.1: Example of Auto-Generated Code (Excerpts)

Recent improvements in programmability of GPUs allow auto-tuning to be applied to GPUs as well. Several CUDA implementations for linear algebra subroutines and FFTs with auto-tuning capability already exist [60, 81, 93].

Previous implementations of stencil computations on GPUs can be grouped into three categories in terms of their emphasis: (1) Hand-coded implementations of a particular stencil strive to achieve the best performance possible [87, 91, 96], but some of their optimization techniques do not even generalize to other types of stencils. (2) Ease of programming is chosen as the primary goal over performance. Such works usually contain code generators for various kind of stencils [43, 115, 84, 72]. (3) Other work focuses on a particular parameter and studies how its tuning can affect performance [82, 86].

We conjecture that performance or programmability are not mutually exclusive. The merit of our work is to offer both ease of programming and performance at the same time. By providing a stencil specification front-end, we alleviate the end-user's burden to master architectural details. Near-optimal performance is achieved by extracting necessary parameters and thoroughly auto-tuning them. Even though some of the aforementioned work utilizes certain tuning parameters, such work either relies on ad-hoc hand tuning [115] or the tuning space is limited [43, 72].



(a) 7-Point (b) 13-Point (c) 19-Point (d) 27-Point

Figure 4.2: Stencil Examples

4.3 Design Overview

The stencil computation considered in this work allows point-wise updates according to a sequence of the following equation over a 3D rectangular domain:

$$\begin{aligned} out([i][j][k]) &= \sum_m w_m * in[i \pm I_m][j \pm J_m][k \pm K_m] \\ &+ \sum_l w_l[i][j][k] * in[i \pm I_l][j \pm J_l][k \pm K_l] \\ &+ \sum_n w_n * in_n \end{aligned} \tag{4.1}$$

The three dimensional addressing in the parenthesis on the left hand side is optional. If absent, we assume the result (out in this case) is an intermediate result that will be used later in another instruction on the right hand side as an input in_n. The first two parts on the right hand side characterize the stencil behavior. The center point and a number of neighboring points in the input grid (in) are weighted by either scalar constants (w_m) or elements in grid variables ($w_l[i][j][k]$) at the same location as the output. Offsets ($I/J/K_m$ and $I/J/K_l$) that constrain how the input grid is accessed are all constant. We call their maxima the halo margins of three dimensions ($halo\_i = \max\{I_{m/l}\}$, $halo\_j = \max\{J_{m/l}\}$ and $halo\_k = \max\{K_{m/l}\}$). To ensure that the access pattern is legal (non-negative indexing) for marginal elements in the input grid in, we assume both input and output grids (in and out) are enlarged by twice the halo margins on each associated dimension.

We differentiate w_ls and in in (4.1) and call them *array parameters* and *array input*, respectively. *Array parameters* are restricted by their access pattern: they can only be accessed at the same position as the output element. The *array input* can be accessed with various constant offsets ($i/j/k$s) on each dimension. We assume there is only one array input, but there can be zero or multiple array parameters.

Given the stencil specification that contains only a list of instructions in the format of Eq. 4.1, our auto-tuning framework generates a header file and an implementation file that can be either included in user code or compiled into libraries.

Excerpts of the generated code are depicted in Figure 4.1. The two major APIs are stencilIteration() and stencilIteration_mpi(). One performs single GPU calculations, the other is for multiple-node GPUs (GPU clusters) computations with node-to-node MPI message passing.

We call a stencil calculation an N-point stencil where N is the total number of input points used to calculate one output point and an order-M stencil where M is the maximum over all $halo_i/j/k$s. In this work, we choose four types of stencil computations as benchmarks (see Figure 4.2).

- 7-Point Stencil (Figure 4.2(a)): Each element in the output grid is updated by the same position in the input grid and 6 neighbors offset by 1 on each direction. The grid point and 6 neighbors are scaled by α and β, respectively, before they are added to generate the output. Both α and β are constants. There are 8 floating-point operations for each point (6 adds and 2 multiplies).

Table 4.1: Specifications of Four Stencil Benchmarks. Indices are subscripted to save space.

Kernel	Specification	# array params	Flops per stencil	mem. refs per stencil
7-point order-1	$tmp = (u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k} + u_{i,j,k+1} + u_{i,j,k-1}) * beta;$ $u1_{i,j,k} = tmp + alpha * u_{i,j,k};$	0	8	8
13-point order-2	$tmp = coef1 * (u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k} + u_{i,j,k+1} + u_{i,j,k-1});$ $tmp+ = coef2 * (u_{i+2,j,k} + u_{i-2,j,k} + u_{i,j+2,k} + u_{i,j-2,k} + u_{i,j,k+2} + u_{i,j,k-2});$ $u1_{i,j,k} = tmp + coef0 * u_{i,j,k};$	0	15	14
19-point order-1 (himeno)	$s0 = wrk1_{i,j,k} + a0d_{i,j,k} * p_{i,j,k+1} + a1d_{i,j,k} * p_{i,j+1,k+1};$ $s0+ = b0d_{i,j,k} * (p_{i,j+1,k+1} - p_{i,j-1,k+1} - p_{i,j+1,k-1} + p_{i,j-1,k-1})$ $+a2d_{i,j,k} * p_{i+1,j,k};$ $s0+ = b1d_{i,j,k} * (p_{i+1,j+1,k} - p_{i-1,j+1,k} - p_{i+1,j-1,k} + p_{i-1,j-1,k});$ $s0+ = b2d_{i,j,k} * (p_{i+1,j,k+1} - p_{i-1,j,k+1} - p_{i+1,j,k-1} + p_{i-1,j,k-1})$ $+c0d_{i,j,k} * p_{i,j,k-1};$ $s0+ = c1d_{i,j,k} * p_{i,j-1,k} + c2d_{i,j,k} * p_{i-1,j,k};$ $ss = (s0 * a3d_{i,j,k} - p_{i,j,k}) * bnd_{i,j,k};$ $wrk2_{i,j,k} = p_{i,j,k} + omega * ss;$	12	32	32
27-point order-1	$b_{i,j,k} = param0 * a_{i,j,k}$ $+param1 * (a_{i-1,j,k} + a_{i+1,j,k} + a_{i,j-1,k} + a_{i,j+1,k} + a_{i,j,k-1} + a_{i,j,k+1})$ $+param2 * (a_{i-1,j-1,k} + a_{i-1,j+1,k} + a_{i+1,j-1,k} + a_{i+1,j+1,k}$ $+a_{i-1,j,k-1} + a_{i-1,j,k+1} + a_{i+1,j,k-1} + a_{i+1,j,k+1}$ $+a_{i,j-1,k-1} + a_{i,j-1,k+1} + a_{i,j+1,k-1} + a_{i,j+1,k+1})$ $+param3 * (a_{i-1,j-1,k-1} + a_{i-1,j-1,k+1} + a_{i-1,j+1,k-1} + a_{i-1,j+1,k+1}$ $+a_{i+1,j-1,k-1} + a_{i+1,j-1,k+1} + a_{i+1,j+1,k-1} + a_{i+1,j+1,k+1});$	0	30	28

- 13-Point Stencil (Figure 4.2(b)): The access pattern resembles the 7-point stencil except that the maximal distance to the neighbors extends to 2, making it an order-2 stencil. There are 15 floating-point operations at each point (12 adds and 3 multiplies).

- 19-Point Stencil (Figure 4.2(c)): This is also called the Himeno benchmark, the behavior of which is detailed elsewhere [96]. We use the same specification (Table I in [96]), except for ignoring the last line of residual calculation. All the weights in this benchmark are array parameters, making it a very cache-unfriendly benchmark. The total number of floating-point operations is 32 and there are 14 memory accesses per point.

- 27-Point Stencil (Figure 4.2(d)): Each grid point computation involves all points in a $3 \times 3 \times 3$ cube surrounding the center grid point. The 4 edge points, 8 corner points and 12 face neighbor points are multiplied by different constants. The number of operations is 30 with 4 multiplies and 26 adds.

Table 4.1 summaries the specifications and properties of the four stencils above.

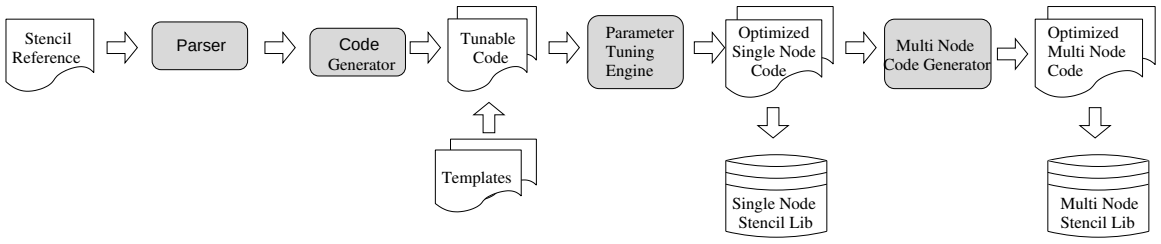


Figure 4.3: System work flow: A user-defined specification is parsed to generate tunable code based on a template. The code is passed to an auto-tuning system to find the best parameter configuration for a single GPU (also for GPU clusters with MPI)

4.3.1 Domain Specification and Framework

The formulation of a stencil is trivial in our framework as users provides a file specifying an equation according to the format of Eq. 4.1 plus parameters, such as the size of each dimension and data type (float or double). Table 4.1 shows that each stencil can be expressed by no more than a few lines of code. In contrast to hand-written CUDA kernels, which usually are a hundreds of lines of code, this is a considerable improvement in terms of productivity. The internal work flow of the framework is depicted in Figure 4.3. The parser analyzes the specification code in terms of Eq. 4.1 and extracts stencil features. These include halo margins ($halo\_i/j/k$), input/output array names, scalar or array parameters (ws) and the number of floating-point operations per stencil. The parser also detects whether the stencil access pattern includes corner element accesses or not. 7-point and 13-point stencils are corner access free because at most one dimensional offset exists when accessing the input array. The code generator takes those feature parameters and chooses different template files according to the corner access pattern before generating tunable code. The auto-tuning engine mainly operates on a single-node level, where optimized parameters are determined based on run-time profiling. The same optimized parameters are used on multiple nodes to generate GPU cluster code with MPI support.

4.3.2 Domain Kernel Template

The design of the template kernel file is affected by the strategy to break the 3D rectangular space into thread blocks in CUDA. In related work, the 3D $X \times Y \times Z$ space was divided into smaller cuboids of size $x \times y \times z$ [43, 85]. Each of them was mapped to a thread block of the same size. Recently, a 2.5D decomposition method was proposed [96, 91]. It decomposes the 3D stencil space over the two most frequently changed dimensions (X and Y). Stencils of size $x \times y \times Z$ are assigned to a thread block, which contains only a plane of $x \times y$ threads. Inside the kernel, threads sweep over the Z axis and cooperatively process one plane at a time.

The benefits of the second method are three-fold: (1) It reduces the pressure on shared memory

usage. In 3D decomposition, each block maintains a small block of size $(x + 2 \times halo\_i) \times (y + 2 \times halo\_j) \times (z + 2 \times halo\_k)$ in shared memory. The 2.5D method only needs a blocks size of $(x + 2 \times halo\_i) \times (y + 2 \times halo\_j) \times (1 + 2 \times halo\_k)$. While sweeping through the z-axis, the planes can be shifted and reused as the work on z-axis is progressed. If the stencil does not have corner accesses, such as 7-point and 13-point stencils, we can further reduce the shared-memory usage to $(x + 2 \times halo\_i) \times (y + 2 \times halo\_j)$ while keeping the other parameters in registers. (2) The 3D decomposition method consumes more memory bandwidth on the Z axis because halo regions on Z are loaded twice on different blocks along the Z axis. (3) The 2.5D decomposition method tends to allocate more stencil points per thread (Z points per thread instead of z points). This is an optimization technique also known as thread fusion. For a large enough problem size, *i.e.*, $(X \times Y)$ generates enough threads, this helps to amortize other overheads, such as initial setup code in the kernel.

In our design, we adopt the block partition strategy in the 2.5D blocking method, *i.e.*, stencil space is partitioned into columns (Figure 4.4(a)). The cross section of each column is of size ($BlockSize.x \times BlockSize.y$), see Figure 4.4(b). We further unroll over both X and Y dimensions to use ($BlockDim.x \times BlockDim.y$) threads per kernel block (see Figure 4.4(c)). Previous work only exploits the unrolling factor at most over the Y dimension. Our experiments illustrate that unrolling over both dimensions can be beneficial (see Section 5.6).

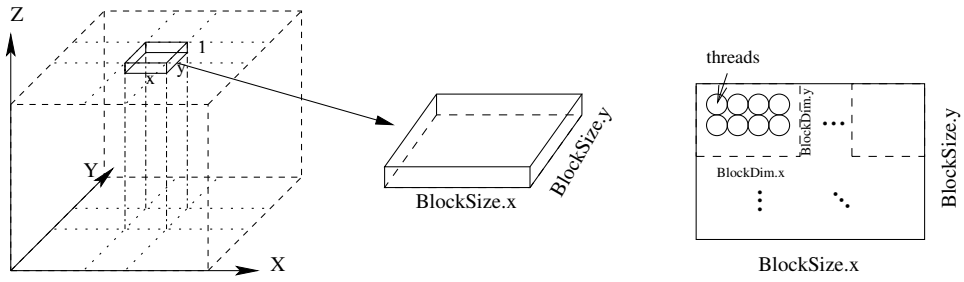


Figure 4.4: Stencil space decomposed over X & Y; process one column per thread block; thread code is unrolled.

Our code generator is based on two kernel templates, depending on whether the stencil has corner accesses (Fig. 4.5(a)) or not (Fig. 4.5(b)), where $halo\_k = 1$ is assumed in these figures. Their most distinct difference is how the shared memory is used. For stencils with corner accesses, all input stencils are first stored in shared memory to calculate the output stencils. The corner-free stencils can be treated as a special case where a plane of stencils does not share inputs other than the points on the same plane.

Therefore, only the middle plane is stored in shared memory in this case — all other inputs along the Z axis are stored in register files. This approach, tailored to corner-free stencils, not only reduces the shared memory pressure but also speeds up the stencil calculation due to the performance advantage of using registers over shared memory.

```
#define sizey (BLOCK_Y+halo_j*2)
#define sizex (BLOCK_X+halo_i*2)

template <class T>
__kernel__ stencil_iteration(...)
{
  // Initialization Instructions
  g_tx = ...;  g_ty = ...;
  ...
  __shared__ T shArr[3][sizey][sizex];
  first = 0; second = 1; third = 2;
  // Load first 2 planes
  shArr[0][][] = ;
  shArr[1][][] = ;
  for (k=halo_k; k<=zSize; k++) {
    // Load third plane to __shared__
    shArr[2][][] = ;
    __syncthreads();
    if (inside) {
     // stencil calculation
      ...
    }
    __syncthreads();
    // Shift planes
    first = (first+1)%3;
    second = (second+1)%3;
    third = (third+1)%3;
} }
```

(a) With Corner Accesses

```
#define sizey (BLOCK_Y+halo_j*2)
#define sizex (BLOCK_X+halo_i*2)

template <class T>
__kernel__ stencil_no_corner(...)
{
  // Initialization Instructions
  g_tx = ...;  g_ty = ...;
  ...
  __shared__ T shArr[sizey][sizex];

  // Load first 2 planes to registers
  T middle = ...;  T below = ...;
  for (k=halo_k; k<=zSize-halo_k; k++) {
    // Shift registers
    top = middle;
    middle = below;
    // load third plane to registers
    T below = ...;
    __syncthreads();
    // load middle plane to __shared__
    ...
    __syncthreads();
    if (inside) {
      // stencil calculation
      ...
} } }
```

(b) Without Corner Accesses

Figure 4.5: Stencil Kernel Templates

4.4 GPU-Specific Auto-Tuning

In the following, we describe in detail various optimization techniques used by our implementation. We reason about their effects on performance and consider if they need to be made elastic by promoting them as parameters for auto-tuning.

4.4.1 Single Node Optimizations

Coalescing Memory Accesses: For NVIDIA GPUs, the latency of global memory references is deeply affected by whether the memory is accessed in coalesced way or not. More recent GPUs support coa-

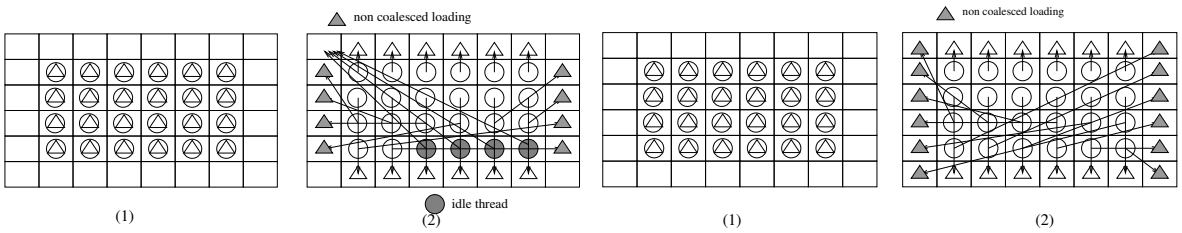
lesced memory access when memory accesses conducted by threads in one warp can be combined into as few memory transactions as possible [7], where a warp is the basic thread instruction scheduling unit in NVIDIA GPUs. We reinforce the following rules to coalesce most of the memory accesses:

- The size of the most frequently changing dimension (X dimension) for input/output arrays is padded to multiples of 32 stencil elements.
- The origin of the input/output arrays are shifted right by $32 - HALO\_I$ stencil elements relative to the memory pointer obtained from the CUDA malloc function. This guarantees 128-bit alignment. The internal origins of the input/output array thus become 128-bit aligned ensuring coalesced memory accesses for output arrays as long as every thread loads the same row at the same time when operating on a half-warp granularity.
- Parameter arrays are allocated to be the same size as the input/output array, even though only the internal elements are used throughout the stencil calculation. This way, the indices of parameter arrays and parameter input become identical saving registers and extra cycles for address calculations. Similar to the input/output arrays, their origins are also shifted to the right. Reading from the parameter arrays become coalesced as well.

Tuning the Block Size: Choosing the right block size is one of the most important factors to balance the utilization of registers and shared memory. Since we use Z-axis sweeps, our blocks have two dimensions of size $BlockSize.x \times BlockSize.y$. The optimal blocking size is determined by several seemingly conflicting factors:

- Since accesses to part of the halo margins are non-coalesced memory accesses, we want to limit these as much as possible. This gives us incentive to increase $BlockSize.x$ as much as possible.
- To reduce the redundant loading of halo margins between different blocks, we need to keep the block close to a square shape.
- The shared-memory usage is proportional to $BlockSize.x \times BlockSize.y$. It must not surpass the shared-memory size on-chip.

Our experiments show that the optimal blocking size can be different under different scenarios: On one hand, different GPU models require different sizes for the same stencil problem. On the other hand, the same GPU model requires different blocking sizes for different stencil problems. To obtain the coalesced memory access effects for an input array, our search space for $BlockSize.x$ is a multiple of the half-warp (16, 32, 48, 64). $BlockSize.y$ has no such constraints. So we sweep its value continuously from 2 to 16. The search space for the CUDA block size ($BlockDim.x$ and $BlockDim.y$) is a subset of the block size search space, with the constraint that $BlockSize.x/y$ is integer divisible by $BlockDim.x/y$. The motivation behind this ratio is that a smaller set of threads has a higher efficiency in using registers. This thorough search gives us the opportunity to balance register utilization and shared memory space, two key resources for stencil implementations on GPUs that are scarce.



(a) No Corner Case (b) With Corner Case

Figure 4.6: Load input sub-plane to shared memory. Internal regions are loaded in Step (1). There is a one-to-one mapping between computing threads and internal regions. In Step (2), the mapping is auto-generated by the parameter tuning engine. (A circle denotes a thread. A triangle denotes an array element loaded at the current step.)

Loading the Input Array Efficiently: An important step in the stencil kernel is to efficiently access the input array. A straightforward but naive implementation is to load it directly from the off-chip global memory while calculating the output point. The obvious drawback is that this does not exploit the data sharing between neighboring threads. The on-chip shared memory serves as an ideal user-controlled scratch pad in this scenario. The problem narrows down to how to efficiently load a larger block of data ($(BlockSize.x + 2 * HALO\_I) \times (BlockSize.y + 2 * HALO\_J)$) using a smaller set of computation threads ($BlockDim.x \times BlockDim.y$). We first load the internal region ($BlockSize.x \times BlockSize.y$). Because $BlockDim.x/y$ are divisible by $BlockSize.x/y$, this can be done easily without branches. For marginal regions, we rely on the code generator to map computational threads to elements on the margin region, as shown in Figure 4.6. In the graph, we assume $BlockDim.x/y$ equals to $BlockSize.x/y$, respectively. Each computing thread is sequentially assigned to a point in the margin area. The x and y indices are auto generated as a constant array. The number of points in the margin area is not necessarily divisible by the number of computing threads. In those cases, threads will be responsible for loading more than one marginal points or there will be idle threads that load the upper-left corner point (see the Figure 4.6(a)) to avoid diverging branches. Comparing with other approaches, e.g., [96], this method neither requires branches nor issues any unnecessary loads. The only non-coalesced memory loads are issued for the columns on each side of the sub-plane.

Using Texture Memory: Mapping the read-only input array into the GPU's texture memory has been shown to improve performance in [96], especially for bandwidth-limited benchmarks. There is no texture support for the double precision data type, but we can use the texture fetch for the int2 type and $\_\_hiloint2double$ to convert it to double. Whether or not to use texture memory for the input array is determined by a boolean tuning parameter.

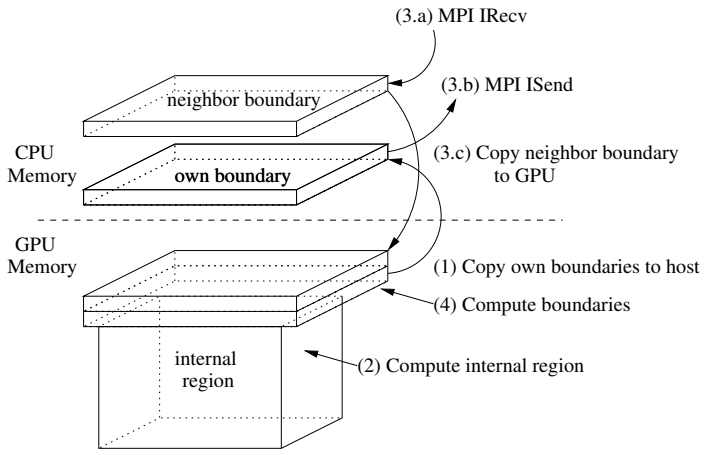


Figure 4.7: Steps in multi-node scenario. For clarity, only one boundary plane is shown.

4.4.2 Multi-Node Auto-Tuning

For GPU clusters, we divide the stencil space along the Cartesian space. Each node is responsible for updating a smaller rectangular 3D space. The tuning parameters determined for a single node are reused directly for multi-node scenarios. However, the code generator needs to break the single kernel into several smaller ones, each of which only processes a portion of the data set. The objective is to separate the six plane boundaries from the internal region. While the boundaries need to be exchanged between neighboring nodes, the internal regions can be calculated completely in parallel with communication.

Our framework generates MPI calls for inter-node communication. At each iteration, each node performs the following steps:

(1) Kernels copy non-continuous boundaries residing in GPU memory into continuous GPU memory buffers. For stencils with corner accesses, eight corners and 12 edges are also copied into separate buffers. Then, continuous boundaries are transferred from GPU memory to host memory via cudaMemcpy.

(2) An asynchronously kernel updates internal regions.

(3) MPI sends and receives are issued to exchange boundaries. Once boundaries are received, boundaries are copied from host memory to GPU memory. This step can be overlapped with the one.

(4) Kernels update stencils on boundaries.

These steps are illustrated in Figure **??**.

For GPU clusters, it is important to keep load balance across all computing nodes. Our Cartesian partition strategy makes sure every node receives nearly the same amount of data to process for a homogeneous GPU cluster. But if we applied the same equal-space partitioning for heterogeneous GPU clusters, more powerful GPUs would finish the calculation first and then wait for slower GPUs for each

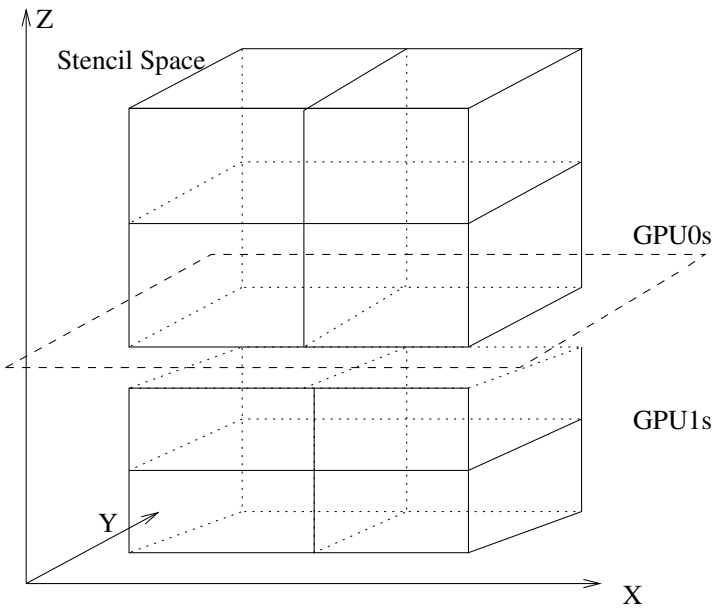


Figure 4.8: Partition stencil space across different GPU types (There are two types in the graph). The space along Z-axis is assigned to GPUs according to their GFlops capabilities.

stencil iteration. Since we already obtained the GFlops rate of a code in the single node auto-tuning step, we can reuse this information for balancing partition sizes. The idea is to partition the data space into layers and assign them to groups of GPUs, where *each group* consists of GPUs with identical models. We choose the least frequently changed axis (Z-axis) to create this partition layer. All GPU groups use the same partition across the X- and Y-axes. The partition lengths on Z-axis are selected proportional to each GPU model's GFlops capability. Here, we assume that the numbers of each GPU model are integer divisible. This guarantees that planes partition the layers. In effect, the communication pattern is the same as that of a homogeneous GPU cluster. Figure **??** shows how partitioning is performed for a cluster with two types of GPUs. The top GPU group has more computational power. Its GPUs are therefore assigned to a larger data space.

4.5 Experimental Results

4.5.1 Experimental Setup

We conducted experiments on single nodes with four NVIDIA GPU models: Geforce GTX 280, Tesla C1060, Tesla C2050 and Geforce GTX 480, spanning two generations of NVIDIA GPUs ranging from

[1]Register File
[2]Shared Memory

Table 4.2: Single Node Experiment Platforms

Model	SM Count	Core Count	L1 Cache	Bandwidth	RF [1]	SM [2]	SP GFlops	DP GFlops
Geforce GTX 280	30	240	N	141.7 GB/s	16 KB	16KB	933	78
Tesla C1060	30	240	N	102.4 GB/s	16 KB	16KB	933	78
Tesla C2050	14	448	Y	144 GB/s	32 KB	16/48 KB	1288	515
Geforce GTX 480	15	480	Y	177.4 GB/s	32 KB	16/48 KB	1345	168

consumer-end graphics card to high performance computing GPUs. Their major specifications are listed in Table 4.2. All kernels are compiled under CUDA 3.2 at O3 optimization level. Experiments with Tesla C2050 are conducted with ECC turned off. For Fermi GPUs (Tesla C2050 and GTX 480), we prefer shared memory over L1 cache since the shared memory size is 48 KB (in contrast to 16 KB in earlier GPUs).

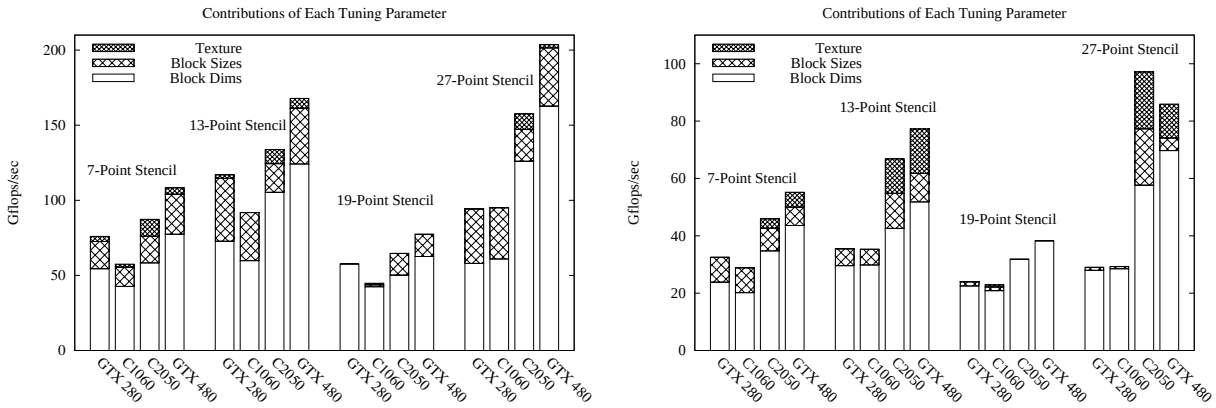
We conducted multi-node experiments on two set of homogeneous GPU clusters connected by QDR Infiniband (36 Gbps) with fat-tree topology. One cluster was comprised of 32 nodes, each with one Tesla C2050, the other had 48 nodes, each with one Geforce GTX480. We further combined the above two GPU clusters to form a larger heterogeneous GPU cluster and apply our hybrid partition strategies.

4.5.2 Single Node Results

Our single-node auto-tuning engine finds the optimal parameters for all stencil types on each GPU model within the given search space. These parameters are shown in Table 4.3. Each GPU model has different optimal settings for all stencil types, even within the same GPU generation. Almost all models favor large $BlockSize.x$ except for some cases with early generation GPUs. These older GPUs

Table 4.3: 7/13/19/27-Point Stencil Results on Single GPU for Single/Double Precision (SP/DP)

Model	BlockSize.x	BlockSize.y	BlockDim.x	BlockDim.y	Texture	**SP GFlops**
Geforce GTX 280	64/32/64/16	8/8/3/6	32/32/64/16	8/2/3/2	Y/Y/N/N	76.0/117.0/57.6/94.2
Tesla C1060	64/64/64/32	8/6/6/8	32/64/64/32	8/2/3/2	Y/N/Y/N	57.5/91.8/44.8/95.5
Tesla C2050	64/64/64/64	8/6/3/4	32/64/32/32	8/3/3/4	Y/Y/N/Y	87.3/133.8/64.6/157.6
Geforce GTX 480	64/64/64/64	3/3/3/8	32/32/32/32	3/3/3/4	Y/Y/N/Y	108.2/167.8/77.4/203.7
Model	BlockSize.x	BlockSize.y	BlockDim.x	BlockDim.y	Texture	**DP GFlops**
Geforce GTX 280	16/16/16/16	16/16/6/6	16/16/16/16	4/8/3/3	N/N/Y/N	32.5/35.4/24.0/29.0
Tesla C1060	32/16/32/16	6/16/4/6	32/16/32/16	2/8/2/3	N/N/Y/N	28.8/35.3/22.8/29.3
Tesla C2050	64/32/64/32	8/6/3/6	32/32/64/32	4/2/3/2	Y/Y/N/Y	45.9/66.8/31.8/97.7
Geforce GTX 480	64/32/64/32	6/6/3/4	32/32/64/16	3/2/3/4	Y/Y/N/Y	55.2/77.2/38.7/86.0

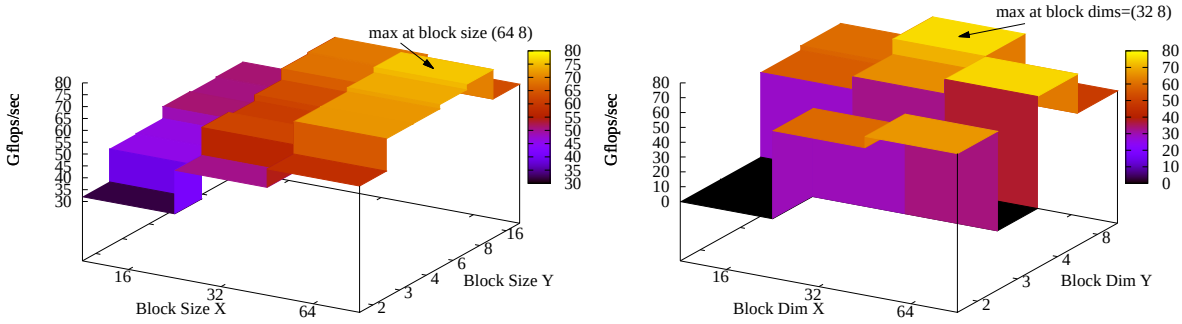


(a) Single Precision (SP) Stencils

(b) Double Precision (DP) Stencils

Figure 4.9: Stencil Tuning Effect Breakups

have tighter restrictions on shared memory size, especially for double precision (DP) stencils. Thus, they can only afford smaller $BlockSize.x$ sizes. $BlockSize.y$ is usually less than $BlockSize.x$, except for 7/13-point DP stencils on a GTX 280 and the 13-point DP stencil on a Tesla C1060 because their smaller $BlockSize.x$ (16) allows them to have a larger $BlockSize.y$. Thus, reducing the non-coalesced memory access (increasing $BlockSize.x$) is favored over reducing redundant loads (increasing $BlockSize.y$).

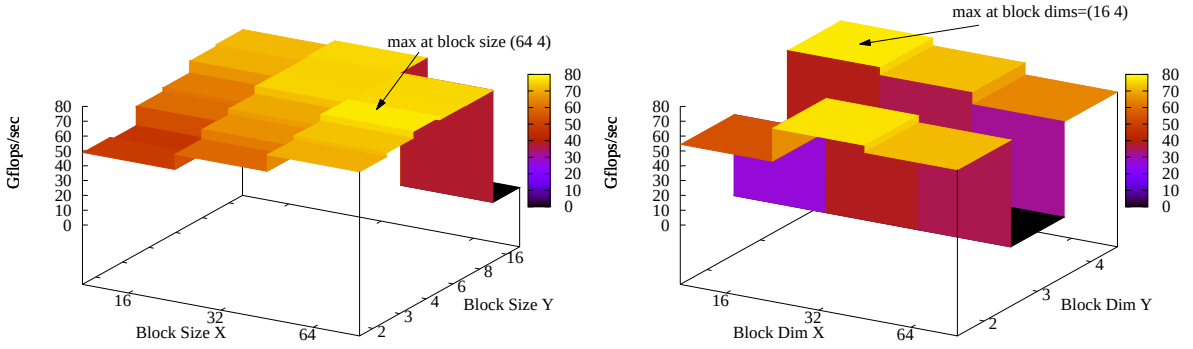


(a) Find Optimal Block Sizes

(b) Find Optimal Block Dims, Fixed Block Size at (64, 8)

Figure 4.10: GTX 280 7-Point Stencil (SP)

An illustration of each tuning parameter's contribution to performance is given in Figure 4.9. Here, auto-tuning is comprised of three steps: (1) $BlockSize.x/y$ are set to be equal to $BlockDim.x/y$; (2) $BlockSizes.x/y$ are tuned for better performance; (3) texture mapping is enabled/disabled. The



(a) Find Optimal Block Sizes

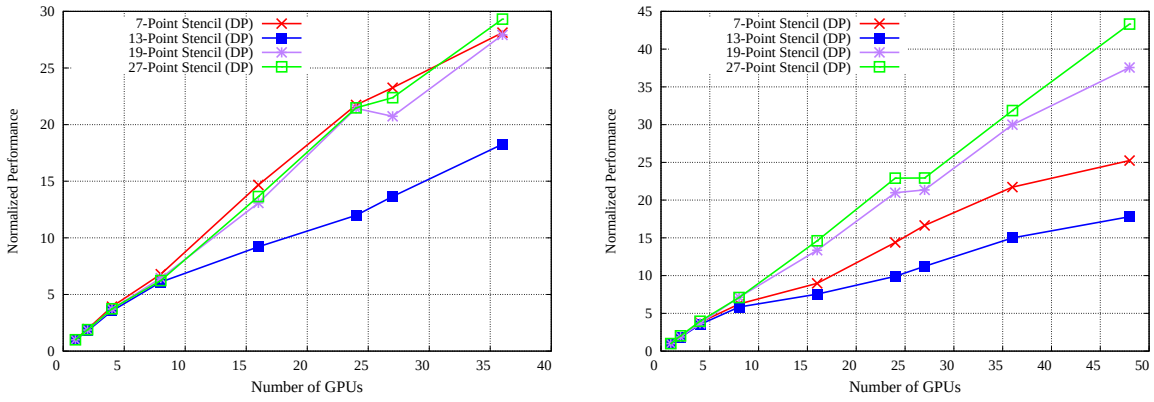
(b) Find Optimal Block Dims, Fixed Block Size at (64, 4)

Figure 4.11: C2050 27-Point Stencil (DP)

necessity to unroll is confirmed by the fact that $BlockDim.x/y$ sizes are almost always different than $BlockSize.x/y$. The only exception is given by a 19-point DP stencil for Fermi GPUs. In this cases, $BlockSize.y$ is too small to unroll. In addition, Fermi GPUs provide enough registers to support a $BlockDim.x$ of the same size as $BlockSize.x$.

Another interesting observation is that mapping the input array to texture memory does not necessarily result in better performance. This is in part because some stencils are not bandwidth-limited on certain GPUs. For GPUs that have high GFlops capabilities, using texture memory usually helps because memory references are on the critical path (7/13/27-point DP stencils for C2050 and GTX 480). Using texture memory has one overhead though: Texture mapping requires the device memory to start from 128-bit aligned address. But our input/output array base addresses are shifted to non-aligned addresses so that the addresses with offset at $halo\_i$ (base address for internal region) are 128-bit aligned. Therefore, there is an extra offset adjustment calculation if we want to enable texture mapping. This extra arithmetic for address computation can negate the benefit of lower latencies for texture memory accesses for some cases.

Of the four GPU models, both Geforce GTX 280 and Tesla C1060 belong to the first generation of CUDA-enabled Nvidia GPUs (computing capability 1.x) while Tesla 2050 and GTX 480 are of the second generation, known as the Fermi architecture. The major difference within a generation is their theoretical memory bandwidth as well as DP performance (for Teslas), which lower models either lack (first generation) or only provide at a lower rate (second generation). Our GFlops rates give us insight to whether a stencil type is bandwidth-limited or computation-limited on a certain GPU. For SP stencils, GFlops rates for Teslas are almost always inferior to that of Geforce models in the same generation, except for the first generation 27-point case. And their ratio is similar to the bandwidth ratio. Therefore, Tesla models are bandwidth-limited in almost all SP stencils. In DP stencils, a similar ratio can be found for 7-point stencil for first generation and 7/13/19-point stencils for second generation GPUs. But for



(a) C2050 cluster

(b) GTX 480 cluster

Figure 4.12: Weak Scaling of DP Stencils on GPU Clusters

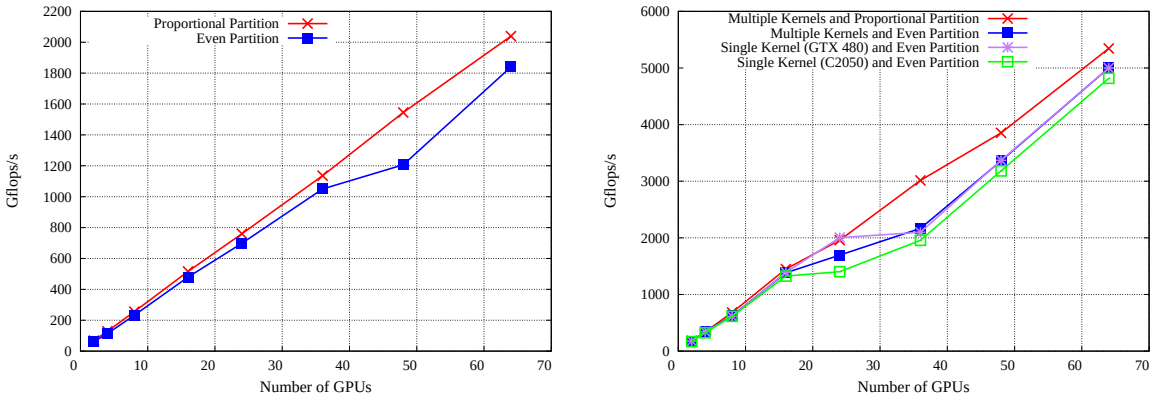
other cases, GFlops rates for Tesla models are close to or better than for Geforce models. Therefore, those stencils are computation-bounded for Geforce GPUs.

To demonstrate the effectiveness of the auto-tuning engine, we select two cases and represent performance in GFlops as a surface in a 3D histogram. Figure 4.10 depicts the single-precision (SP) 7-point stencil on a GTX 280. Figure 4.11 depicts the DP 27-point stencil on a Tesla C2050. The left diagrams in the figures illustrate how the performance changes while varying $BlockSize.x/y$, assuming the best $BlockDim.x/y$ has been found. The right diagrams in the figures depicts how the performance changes when varying $BlockDim.x/y$ for a fixed $BlockSize.x/y$ overall. The figures demonstrate that each tuning parameter plays an important role in the final performance, neither one of which can be explored independently of the other. Hence, an auto-tuner needs to exhaustively test all permutations.

Our auto-tuning engine does exactly that: an exhaustive search over all possible permutations is performed. This guarantees a global optimum with respect to the parameter search space. Adaptive search methods could be adopted to prune the search space. However, care must be taken because local optima exist, as seen in the figures. For example, in Figure 4.10(b), (64,4) is another locally optimal $BlockDim.x/y$ pair. Considering the search space is relatively small (less than 200 combinations in the worst case), exhaustive search is feasible as individual runs can be short.

4.5.3 Multi-Node Results

We study the weak scaling property [61] of our framework in the two GPU clusters. We keep the problem size per GPU constant and increase the stencil size over all three dimensions at roughly the same rate as the increase in number of GPUs. Therefore, the stencil space is kept as close to a cube as possible. The Y axis of Figure 4.12 depicts the normalized performance (measured in GFlops) of a single GPU. For the C2050 GPU cluster, all three order-1 stencils (7/19/27-point) show better efficiency



(a) 19-point DP

(b) 27-point DP

Figure 4.13: Performance Results on Heterogeneous GPU Cluster (The ratio of two GPUs is 1:1 in all experiments)

(77% to 80%) than order-2 stencil (50% for 13-point). Because the GTX 480 has higher single-node DP GFlops for 7/13/19-point stencils, the weak scaling efficiency is worse than that on the C2050 cluster. But for 27-point stencils, GTX 480's single-node DP GFlops is less than C2050's DP GFlops. Therefore, the efficiency is better (about 90%). This can be explained by the difference in inter-node message sizes required by different stencils types. The message size is roughly proportional to the degree of the stencil order. Therefore, our 13-point stencil is communication-bound in our current cluster configuration.

Some of the curves do not show a noticeable improvement from 24 to 27 GPUs (nodes). The 19-point stencil curve even shows a slight drop. This is because the stencil space is divided into $2 \times 3 \times 4$ and $3 \times 3 \times 3$ partitions in these two cases, respectively. The latter case contains a center node that needs to communicate with all other 26 nodes. This node becomes a hot-spot and reduces the performance. But as we increase the number of GPUs, the curve recovers to the expected slope for weak scaling.

To demonstrate the effectiveness of auto-tuning and proportional partitioning on heterogeneous GPU clusters, we compare the GFlops in three different setups:

- Multiple Kernels and Proportional Partitioning: We generate separate kernels with the auto-tuned parameters for each GPU type and divide stencil space according to their GFlops capabilities. This is the optimal setup.

- Multiple Kernels and Even Partitioning: We use optimized kernels for each GPU type but evenly divide the stencil space among all GPU types.

- Single Kernel and Even Partitioning: We use just one kernel for all GPU types and evenly divide the stencil space. Since we have two types of GPUs in our cluster, we test two different kernels (for each of the single-GPU optimal parameters), unless they have the same parameter settings.

Figure 4.13 shows the GFlops/s rate versus the number of GPUs in 19-point and 27-point DP stencil code. Because the optimal parameter setting for both GTX480 and C2050 are the same for 19-point stencil (see Table 4.3), they can share the same stencil kernels to achieve the best performance. The last two setups become identical in this case. Figure 4.13(a) demonstrates that proportional partitioning is always superior to even partitioning. The first (optimal) setup produces the best GFlops/s rate among all four curves in Figure 4.13(b). Because a single C2050 performs better than a GTX480 for a 27-point stencil (97.7 GFlops versus 86 GFlops), GTX480s are on the critical path in the heterogeneous cluster. This explains why kernels optimized for GTX480 outperform kernels optimized for C2050. The performance difference diminishes for larger scales in both figures. This can be explained as follows: The larger the scale, the greater is the communication to computation ratio (the message size increases but computation is kept constant here) and the less important it becomes to reduce computation time discrepancies.

4.5.4 Comparison with Previous Work

We report our results on a wide range of GPUs and stencil types, which allows us to compare our performance directly with a wide range of prior work, both for handwritten and auto-generated codes.

Datta *et al.*'s work on optimizing stencil codes in multi-core architectures including GPUs is one of the early contributions in this area [43]. They showed an unprecedented 36 GFlops for 7-point stencil on a GTX 280 with their highly optimized code. Theirs is 10% faster than our performance (32.5 GFlops). This is mainly due to the difference between the instruction orders in our template file and their hand-tuned kernel code, as we discovered by inspecting their and our codes side-by-side. But interestingly, their best performance is achieved at a block size of 16×16 and unroll factor of 4 over the dimension Y, which is consistent to our findings in our auto-tuning engine. However, this configuration is *only* optimal for a DP 7-point stencil on the GTX 280s. For everything else, the 16×16 block sizes are no longer optimal, as indicated by Table 4.3.

An efficient and handwritten CUDA implementation on the Himeno benchmark is reported by Philips *et al.* [96]. Their implementation, with an extra two Flops per stencil for residual calculation, achieved 50 GFlops SP on a Tesla C1060. Our auto-generated code achieves 44.8 GFlops on the same platform and is within 5% to theirs if Flops are normalized ($44.8 \times \frac{34}{32} = 47.6$). Their best block sizes are 64×2 for Tesla C1060, while ours is 64×6 with an unrolling factor of 2 over the Y axis. This is because they load the input arrays into shared memory by issuing four branch-free loads aligned at four corners. Choosing $BlockSize.y$ as 2, in their case, minimizes redundant memory loads, which is beneficial because SP Himeno is bandwidth limited on the C1060. They also reported near-perfect weak scaling efficiency on up to 16 GPUs. But their system configuration is different from ours: (1) Each node has two GPUs instead of one in our case. Therefore, half of the network messages become memory copies on the same host. (2) The stencil space only grows along the Z axis, eliminating the

need to perform Cartesian partitioning. This reduces the multi-node code complexity significantly.

Kamil *et al.* proposed an auto-tuning framework for multi-core architectures [72]. However, they reported only 14 GFlops DP on a 7-point stencil for a GTX 280. This is mainly because their code generator does not take advantage of the fast on-chip shared memory, which is an ideal intermediate storage level to reduce memory load for stencil-like computations.

Nguyen *et al.* have reported by far the fastest implementation of any SP stencil code on single GPU [91]. Their manually-written code for a 7-point stencil achieves 136 GFlops on GTX 285 (a similar platform as GTX 280), a large gain over our reported 76 GFlops. However, their extra speedup comes from saving a large amount of global memory accesses by exploiting data locality on the time domain. This is equivalent to executing several iterations per kernel, a technique also known as increasing the ghost region. Increasing the ghost region leads to less frequent message exchanges but does not reduce the total amount of data transferred in the network because the payload for each message increases as well. It has been shown to be insignificant in multi-node scenarios due to the slower inter-node communication [104]. Therefore, we decided not to include ghost region sizes/update frequencies as a tuning parameter in our code generator and auto-tuning schemes. For DP stencils, their performance is no better than [43] due to limitations in shared memory size of the GTX 285.

Unat *et al.* proposed a compiler framework called Mint using annotated C as the front-end. It converts stencil computation into C code using pragmas with several levels of optimized CUDA code [115]. Our DP performance of a 7-point stencil on the C1060 achieves the same GFlops as their hand-written code (28 GFlops). In contrast, auto-generated Mint code with the highest level optimization achieves only 22 GFlops.

Christen *et al.* [39] and Maruyama *et al.* [84] proposed two DSLs: Patus and Physis. Patus purely depends on the cache on the Fermi architecture without using any shared memory. Therefore, its auto-tuning capability is severely limited. Physis currently lacks any auto-tuning scheme, one has to choose block sizes manually. Both report SP performance inferior to ours.

4.6 Conclusion

This work shows that GPU programmability and performance are not mutually exclusive under DSLs. With a DSL specification fed to the front-end, problem descriptions can become very concise and intuitive. Using auto-tuning with run-time profile feedback, optimal tuning points within the parameter search space can be identified. Our framework combines auto-generation and auto-tuning of 3D stencil codes on heterogeneous GPU clusters. We extract a small, selective number of key performance-sensitive parameters and auto-tune them to achieve the best possible performance over a variety of GPUs. Compared to previous work, we manage to keep the programmer's effort to even a lower overhead without significant sacrifice in performance. We also show that heterogeneous GPU clusters exhibit the when leveraging *proportional partitioning* of the data space relative to single-GPU performance.

Chapter 5

CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures

5.1 Introduction

Exploiting data parallelism is crucial for programming on many-core architectures because data parallelism exposes a much higher degree of parallelism than task or pipeline parallelism. This high degree of parallelism is necessary to keep up with the ever-increasing instruction throughput provided by hardware. However, popular languages, such as C/C++, do not treat data parallelism as a first-class citizen. This gap between the front-end language and hardware is exacerbated by the fact that compilers are struggling to extract data parallelism from language abstractions. Therefore, human assistance is often necessary to increase performance, which adversely affects the programmer's productivity.

Although adding new ways to pass critical data parallelism information to the compiler (*e.g.,* via pragmas) for task-oriented languages maybe a viable method, we take a completely different approach. We investigate what compiling techniques are needed to efficiently map data-parallel languages to state-of-the-art GPU architectures. One of the advantages of this approach is to improve programming productivity because data-parallel languages are often found to be more concise and elegant to express parallel algorithms.

Among the various data parallel languages, NESL [22] is of particular interest. It is based on the concept of nested data parallel abstractions, which are very common in divide-and-conquer parallel algorithms. An apply-to-each construct encourages programmers to think about algorithms in a parallel fashion at the finest data granularity, thus removing the burden for compilers to engage in complicated data dependence analysis. Recursive calls are widely used in NESL, an elegant way to express nested parallelism.

Previous research has studied how to compile data-parallel languages for SIMD vector machines as well as MIMD parallel machines ([20, 33]). Our work, in contrast, targets the SIMT paradigm, an

increasingly popular programming model advocated by modern GPU architectures. There is a fundamental difference between SIMD and SIMT. In SIMD, only one flow of control exists. The width of vector data that an instruction operates on can swiftly vary from one instruction to another. SIMT, in contrast, has more control flow resources to support many independent threads, each of which can execute instructions asynchronously. The loosely-coupled threading model in SIMT gives programmers more flexibility and removes the lock-step synchronization of SIMD. Generally speaking, it is more challenging to fully utilize the hardware resources in SIMT. SIMT requires coarser grained data parallelism with many, preferably independent entities to achieve good performance.

An important vectorizing compiler technique is the transformation of nested data parallel languages to SIMD code. The transformed code can be ideally mapped into the *Parallel Vector Model*, which contains a vector processor and a "flattened" vector memory [20]. Unfortunately, today's SIMT is not truly "flattened". In particular, CUDA-enabled GPUs consist of hierarchical levels of threading models with different synchronization properties ([7]): (level-1) a host level CPU thread; (level-2) massive numbers of asynchronous threads in CUDA kernels; (level-3) moderate numbers of synchronizable threads in a CUDA kernel block and (level-4) a relatively small number of lock-step synchronized threads in a warp. A naïve execution of the transformed code uses the level-1 CPU thread as flow control and treats the set of level-2 threads as a unified vector processor. Under the CUDA model, explicit barriers are required for each nesting level but are not supported in hardware at level 2. This lack of support results in many unnecessary and expensive global barriers (at level 1) between explicit kernel calls issued by the CPU, which adversely affects performance.

Therefore, it is desirable to delve into the threading model hierarchy and take advantage of low-overhead local synchronizations. To that end, we spawn the control flow at level-3/4. But recursive functions in NESL pose a performance hurdle. In previous approaches, there was no motivation to remove recursions during code transformations. Yet, invoking recursive functions at level-2/3 will cause overhead such as branch penalties and results in imbalanced computation load. Such overhead cannot be neglected and may consume the benefits of faster local synchronizations.

As we can see, previous code transformations no longer suffice to generate code suitable for today's hierarchical SIMT architectures. In this work, we design a source-to-source compiler to directly convert NESL to CUDA code that can be efficiently executed on contemporary NVIDIA GPUs. We focus on recursive NESL functions. In addition to the vectorization transformations, we restructure control flow to remove recursion and provide fine-grained data granularity suitable for SIMT architectures. A recursion-free control flow allows us to dynamically switch between hierarchical threading models and then to choose the best one under different scenarios.

The current CuNesl compiler targets CUDA C++, a vendor proprietary programming model from NVIDIA. However, the proposed compiler techniques can be extended to other data-parallel languages, such as data-parallel Haskell [28].

5.2 NESL Language

In this section, we give a brief introduction to NESL. NESL is an example of a data-parallel language, also known as a collection-oriented language [107]. It is strongly typed and declarative (free of side-effects).

Like other data-parallel languages, NESL consists of standard apply-to-each (map) constructs. The apply-to-each construct applies a certain operation to all elements of a sequence. For example, the expression

```
{negate(a): a in [3, -4, -9, 5]};
```

negates the sequence of numbers in an element-wise fashion in *parallel*, resulting in a sequence of values [-3, 4, 9, 5]. NESL ensures that apply-to-each constructs can be executed independently per element. Therefore, they can easily be mapped onto data-parallel execution models.

A set of primitive parallel functions that can operate on sequences are pre-defined in NESL as well. These functions are not necessarily embarrassingly parallel but still represent efficient parallel algorithms. An example is "b = permute(a,i)", where sequence b is formed in such a way that the *j*th elements in sequence a is permuted to position i[*j*] for all *j*s.

Support for nested parallelism is one of the key ideas behind NESL. Elements in a sequence in NESL can itself be a sequence, which supports recursively nested sequences. Such nested parallelism comes from NESL's ability to apply any function in parallel over the elements of a nested sequence. For example, a sum applied to a nested sequence forms a set of parallel sum calls in a nested fashion.

```
{sum(a) : a in [[2,3], [8,3,9], [7]]};
=> it = [5, 20, 7] : [int]
```

NESL defines several functions to support nesting and unnesting of a sequence, including flattening (reducing the nesting by one level) and bottop (splitting a sequence in two halves and returning them as a nested sequence). NESL is very powerful in expressing divide-and-conquer parallel algorithms with nested recursive calls. Quicksort written in NESL is depicted in Figure 5.1. The expression

```
result = {qsort(v): v in [less, greater]}
```

applies the recursive calls to qsort on a nested sequence formed by less and greater sequences. Nested parallelism, in this case, means that both the two qsorts and the generation of three intermediate arrays inside qsort can be performed in parallel.

5.2.1 Segmented Array

Previous research translates data-parallel languages (*e.g.*, NESL) into a stack-based intermediate language called VCODE ([21]), which is tailored to SIMD machines. This transformation is called *flattening of nested parallelism* [19]. The basic data type of VCODE is a flattened segmented array.

```
1 function qsort(a) =
2 if (#a < 2) then a
3 else
4    let pivot = a[#a/2];
5        less = {e in a| e < pivot};
6        equal = {e in a| e == pivot};
7        greater = {e in a| e > pivot};
8        result = {qsort(v): v in [less,greater]};
9    in result[0] ++ equal ++ result[1] $
```

Figure 5.1: Quicksort in NESL

Figure 5.2 depicts the dynamic partitioning of segmented arrays for quicksort. Each row in the figure is a segmented array. Initially, a single segmented array with just one segment exists. As more and more partitions are formed, the segmented array breaks into many smaller segments.

CuNesl adopts this concept and provides an efficient implementation for pre-defined parallel operations on segmented arrays. We will provide more details in Section 5.5.1.

5.3 Related Work

Programming on SIMT architectures has quickly become mainstream since the launch of CUDA and has changed the GPU's image from that of a purely graphics-specific accelerator to a general-purpose co-processor. While a tremendous numbers of applications can benefit from manually rewriting legacy code for CUDA, many researchers strive to improve the programmability without sacrificing performance.

One approach is to provide handwritten, highly-efficient implementations for well-defined APIs so that they can directly be used by other programs. CUDPP [67], Jacket [68] and Thrust [65] are examples of this approach. In fact, CuNesl's implementation directly uses CUDPP's parallel scan/reduce APIs.

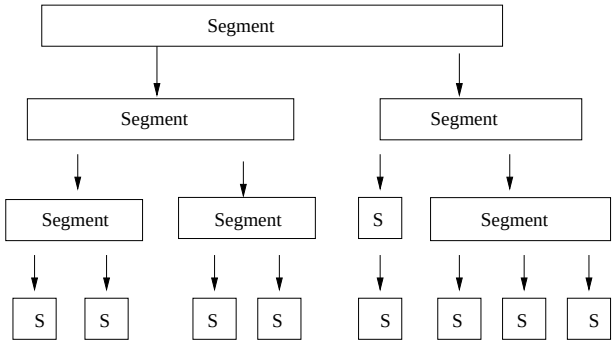


Figure 5.2: Segmented Array in Quicksort: Each row is a segmented array.

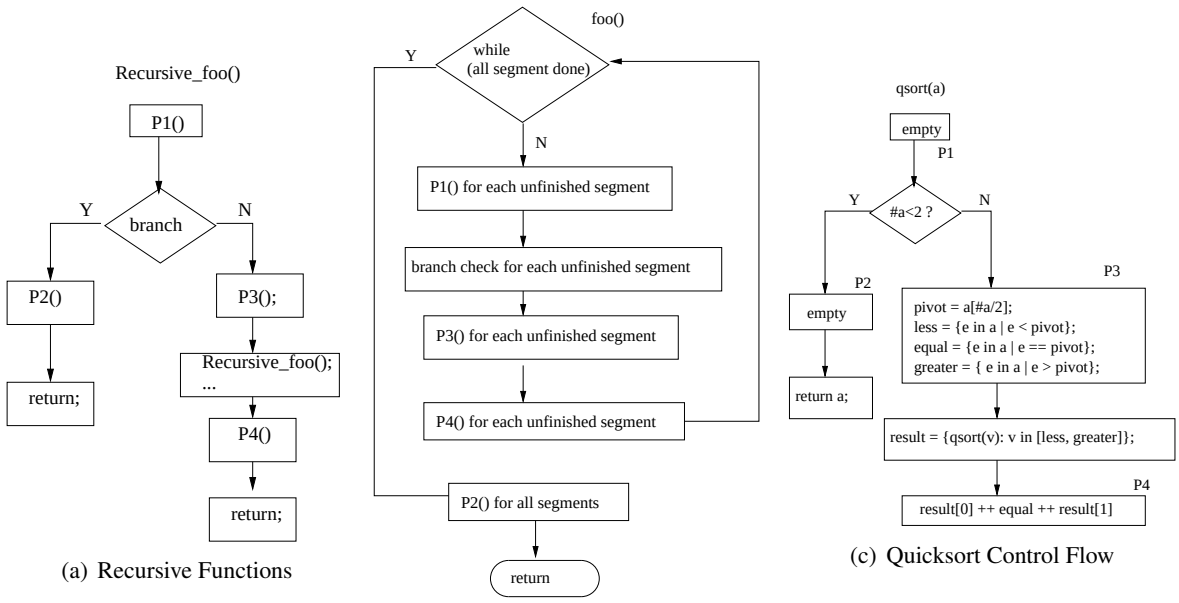
However, this only applies to certain areas where the interface can be clearly defined or standards exist.

By restricting problems into specific domains, compilers can aggressively exploit domain-specific knowledge to auto-generate efficient CUDA code. Domains like stencil computation [115, 84, 39], streaming [114] and PDE solvers [44] are already benefiting from this approach.

For general-purpose languages, a common method is to add directives (*e.g.*, pragmas) to enable code generation by the CUDA back-end. They can be either extending existing directives like OpenMP [79] or introduce new sets of pragmas [62, 122]. There are also source-to-source compilers that translate a naive CUDA kernel into an optimized highly efficient version [121].

In terms of data-parallel languages, the PGI CUDA Fortran Compiler [59] directly compiles HPF into CUDA source code. The compilation of other data-parallel languages, such as Haskell and Python, into CUDA code is still an active research topic [77, 53, 26].

CuNesl shares the same philosophy as Copperhead [26] in that a hierarchical execution model should be exploited in today's architectures to achieve good performance for nested parallelism. CuNesl also extends the applicability of this concept to recursive calls, which cannot easily be statically mapped to finite execution hierarchies and are thus beyond Copperhead. In addition, we show that a *nested* flattening transformation, if coupled with data-flow analysis on the transformed code, matches the hierarchical execution model for SIMT architectures and results in additional performance benefits.



(a) Recursive Functions

(b) Segmented Version

(c) Quicksort Control Flow

Figure 5.3: Convert (a) Recursive_foo() into (b) a recursion-free while loop with (c) an example for Quicksort.

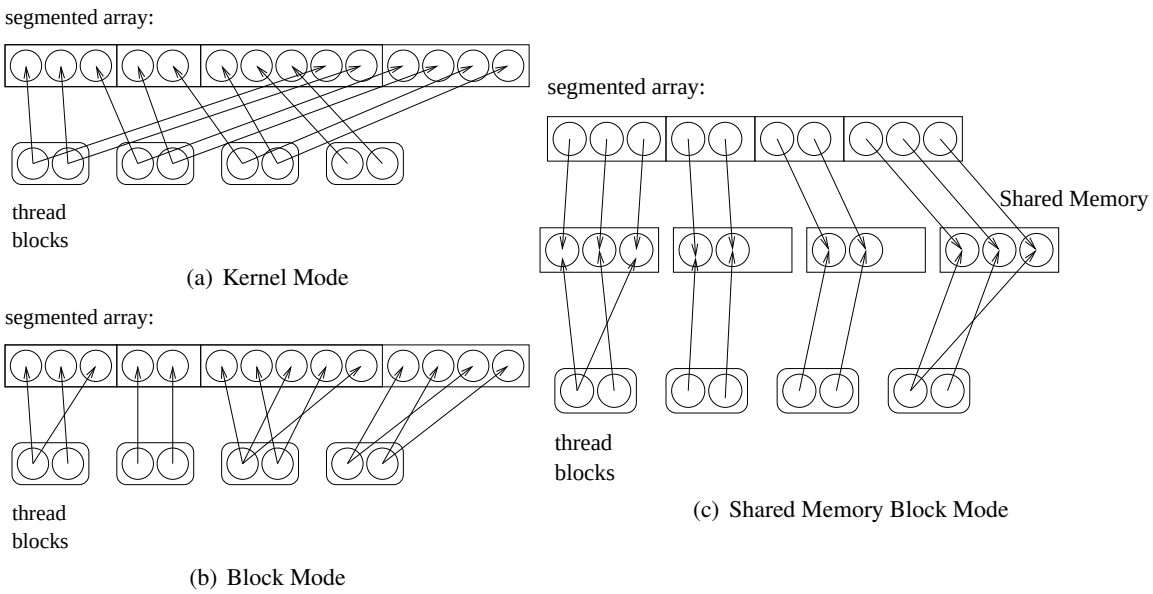


Figure 5.4: Different Execution Modes. In kernel mode, threads process elements in the array globally. In block mode, one block is assigned to a segment. Shared Memory block mode is an optimized version of block mode. It utilizes the on-chip Shared Memory to reduce global memory accesses.

5.4 CuNesl Compiler

5.4.1 Removing Recursive Calls

As discussed in Section 5.2.1, removing the recursive calls in NESL is important for efficient compilation in SIMT architectures. In this section, we will use quicksort as an example to show how CuNesl maps a recursive function into a while loop, even for some non-tail recursion cases.

For a recursive function to terminate, there are always conditional branches inside the recursive function. At least one of the branches does not make further recursive calls. A simplified control flow for a recursive function is illustrated in Figure 5.3(a). The P2() branch is the exit path for the recursive call. In the recursive path, if P4() is empty and this path directly returns after issuing a single recursive call, then it is a tail-recursive function. Most of the NESL examples do not fall into the category of tail-recursion, for they either have multiple recursive calls or have a non-empty P4() block. Fortunately, NESL's syntax guarantees that such multiple recursive calls, if they exist, can be executed in parallel. And it is often the case that P4() is a simple operation that, if positioned prior to the recursive calls, does not affect the final output. Examples of such operations are concatenation, flatten and bottop. Figure 5.3(b) shows the recursion-free transformation based on the above assumptions. The recursive function

```
void qsort(SegmentArray<T> &array) {
  while(!array.isRecursiveAllDone()) {
    /* branch 0, check segment length */
    array.setRecDoneByLength(1);
    /* branch 1, */
    MirroredArray<T> pivots(array.
        getNumSegments());
    gen_pivots(array, pivots);

    MirroredArray<uint> less_flag(array.getSize()
        );
    ...;
    gen_flags_from_pivot(array, pivots, less_flag,
        ...);

    FlagSubIrregularSegmentArray<T> less();
    FlagSubIrregularSegmentArray<T> equal();
    FlagSubIrregularSegmentArray<T> greater();

    FlagSubIrregularSegmentArray<T> *children
        [3];
    children[0] = &lesser;
    children[1] = &equal;
    children[2] = &greater;

    /* reshuffle each segment in array into 3
        segments,
       a built-in function in segmented array */
    array.reshuffle(&children[0], 3);
  }
}
```

(a) Generated Code for Quicksort

```
1  template <class T> __global__ void
2  quicksort_block(IrregularSegmentGpuArrayC<T> *
       array) {
3    __shared__ FlagSubIrregularSegmentGpuArray<T>
       less;
4    __shared__ IrregularSegmentGpuArrayC<T> s_array;
5    __shared__ GpuArray<uint> less_flag; ...
6    __shared__ GpuArray<T> pivots;
7    // temporary buffer for parallel scan/reduce
8    __shared__ uint mSharedBuffer[...];
9    __shared__ FlagSubIrregularSegmentGpuArray<T> *
       children[3];
10
11   __syncthreads();
12   // copy the segment info locally
13   if (threadIdx.x == 0)
14     array->clone(&s_array);
15
16   __syncthreads();
17   ...
18   int segid = blockIdx.x;
19   if (array->isRecursiveDone(segid))
20     return;
21   // prepare for the assigned segment
22   s_array.convertToLocal(bid);
23   __syncthreads();
24
25   // the while loop from the recursive call
26   while (!s_array.isRecursiveAllDone()) {
27     s_array.setRecDoneByLength(1);
28     __syncthreads();
29     if (s_array.isRecursiveAllDone())
30       break;
31     __syncthreads();
32     ...;
33     gen_pivots_block(s_array, pivots);
34     __syncthreads();
35     gen_flags_block(s_array, pivots, less_flag,..);
36     __syncthreads();
37     ...;
38     s_array.reshuffle(children, 3, mSharedBuffer);
39     __syncthreads();
40   }
41   // copy the data back to the global array
42   s_array.copyFromLocal();
43 }
```

(b) Generated Code for Quicksort in Block Mode

Figure 5.5: Generated Code for Quicksort

is now replaced by a while loop, which exits once all segments terminate (have reached the exit branch). Inside the loop, all operations are applied to segments that have not been marked as finished. P2() is

executed when the loop exits. Here, we also assume that P2()'s execution can be safely moved to the end. Otherwise, this step need would need to be moved inside the while loop so that it is applied to every segment that has *just* terminated.

Quicksort in NESL is an example of an algorithm that can be transformed into a parallel tail-recursive function. The mapping from NESL source code into the recursive control flow is shown in 5.3(c). P4() is a simple concatenation operation. Therefore, the compiler can perform code motion to place it before issuing the recursive calls. This is done by inserting the "equal" sequence in in between the "less" and "greater" sequences and marking this as a non-recursive segment.

Figure 5.5(a) lists the resulting code generated for quicksort. The compiler fuses operations that share the same input into one operation. For instance, all three intermediate flag arrays for "less", "equal" and "greater" are generated from the same kernel function. The concatenation of three segments is a built-in function of our segmented array (reshuffle() method).

5.4.2 Hybrid Execution Mode

After converting recursive routines into iterative while loops, we have successfully flattened the program and made it suitable for SIMT architectures. Threads can now start from the bottom and work at the finest data granularity during the entire execution. But in practice, this transformation alone does not usually deliver competitive performance. The reason is that today's SIMT architectures consist of a hierarchy of execution modes, each of which has its own characteristics. Consider CUDA, which has the following execution levels:

Kernel level: This is similar to the bulk synchronization model [116]. Control flow is driven by one or a few host threads on the CPU side. Concurrent computation is performed by launching massively-threaded CUDA kernels. Global synchronization is feasible (between CUDA kernel launches) but relatively expensive.

Block level: This level operates inside CUDA kernels on a GPU. Threads in the same block execute the same program, but do not necessarily proceed at the same rate. Sharing between threads can be realized via Shared Memory. Synchronization at block level is relatively cheap. In the CUDA context, it is supported trough the $\_\_syncthreads()$ API call.

Warp level: This level is similar to SIMD in the sense that threads in the same warp execute programs at the same pace on a GPU. There can be one or more warps at the upper block level. Branches are more efficiently executed if threads in the warp all agree to take the same path. Synchronization between warps is zero-overhead because it is enforced by the hardware via lock-step execution.

Experienced CUDA programmers often choose particular execution levels to solve different problems, or even a problem at different stages, based on various factors. If global synchronization is only occasionally required, a kernel level program should be designed. If a problem can be divided into at least a moderate number of independent smaller problems (a divide-and-conquer approach), it is gener-

ally more efficient to work at the block level because kernel launch overhead and global synchronization are reduced. It is not uncommon to utilize the lock-step synchronization property at the warp level for small but communication-rich operations. Such examples can be found in efficient CUDA implementations of parallel reduce or scan [64].

Lessons learned from coding styles of real-world applications lead us to believe that a hybrid execution mode is necessary to achieve good performance in CuNesl. A truly flattened hardware is not likely to be available due to the unavoidable tradeoff between hardware resources and performance. Relying on a flattened execution mode will only underutilize the hardware, which would make such a method inferior to other approaches.

Therefore, we define several execution modes in CuNesl corresponding to the hierarchical levels of hardware abstractions. This is best explained in the context of how to access and manipulate elements in a segmented array for a massive number of independent SIMT threads. Right now, CuNesl defines the following three execution modes:

Kernel Mode:This mode corresponds to the kernel level abstraction above. When the segmented array consists of only a few large segments, it does not make sense to assign a large segment to only one thread block. Instead, it is more efficient to spawn as many threads as possible and allow multiple blocks to work on the same segment (Figure 5.4(a)). The drawback of this mode is that synchronizations across a segment can only be performed between disjoint CUDA kernels, which is relatively expensive. From the recursive routine's point of view, this mode is usually advocated at the beginning of a recursive call where the number of partitions is small. The foreach operations on a segmented array are translated into kernel pseudo code like the following:

```
stepsize = blockDim.x * gridDim.x;
for (id = threadIdx.x; id < size; id += stepsize)
{
   segid = getSegId(id);
   seglen = getSegLen(segid);
   ...
}
```

Block Mode: When the segment array contains a moderate number of segments, we can assign each segment to an exclusive thread block (Figure 5.4(b)). This corresponds to the block level abstraction. Because a barrier is supported within a thread block, many operations on segments, though not embarrassingly parallel, can be performed without leaving the kernel, thus reducing kernel launch overheads. This mode can often be applied during the mid-phase of a recursive call when enough partitions are produced to fully utilize the many-cores of SIMT architectures. The foreach operations on a segmented array, in this mode, is translated to the following pseudo-code inside the CUDA kernel:

```
stepsize = blockDim.x;
```

```
segid = blockIdx.x;
segsize = getSegLen(segid);
segoffset = getSegOffset(segid);
for (id = threadIdx.x; id < segsize; id += stepsize)
{
  my_global_id = segoffset + id;
  ...
}
```

Shared Memory Block Mode: One important and effective optimization opportunity arises when the size of each segment becomes small enough to fit in the on-chip Shared Memory. We can preload segments into Shared Memory first and work on them before storing them back to global memory (Figure 5.4(c)). This way, we can reduce memory bandwidth consumption. Because Shared Memory is limited in size, this mode is usually feasible and more efficient near the end of a recursive call. This mode can be regarded as an optimized version of *block mode*.

As of now, we have not explored the benefits of going down to the warp level in CuNesl because warp level programming is often found in low-level libraries that are *used* by CuNesl. This is not to say that the lockstep synchronization at warp level is unimportant. A study to assess if this mode is beneficial for more general cases is subject to future research.

Going back to quicksort, the pseudocode in Figure 5.5(a) is in fact generated for the *kernel mode*, only. To switch to other execution modes, CuNesl adds a counter check inside the while loop to exit the loop early, i.e., once the number of segments exceeds a threshold. It then calls a single CUDA kernel that executes the rest of the iterations in *block mode*. The pseudo-code for this single kernel is shown in Figure 5.5(b). It contains a similar while loop as in the *kernel mode* (Figure 5.5(a)). Functions that are used as kernel calls in *kernel mode* are transformed into device functions in a segmented version. Barrier synchronization is provided by __syncthreads() between parallel regions.

5.5 Runtime

5.5.1 Segmented Array

The core of CuNesl's runtime system supports the necessary primitives for segmented arrays. Segmented arrays are encapsulated in various classes that can be included in the compiler-emitted code. They are further compiled by NVCC to generate binaries. To support the concept of a segmented array, to make it conveniently available to the programmer and to ensure efficiency for fine-grained SIMT threads to work on individual elements, we add several auxiliary arrays besides the raw data array to maintain the state of a segmented array (assuming its size is N):

mSegments: array of size N. Elements in this array are either 1 or 0. A 1 indicates the start of a new segmented array.

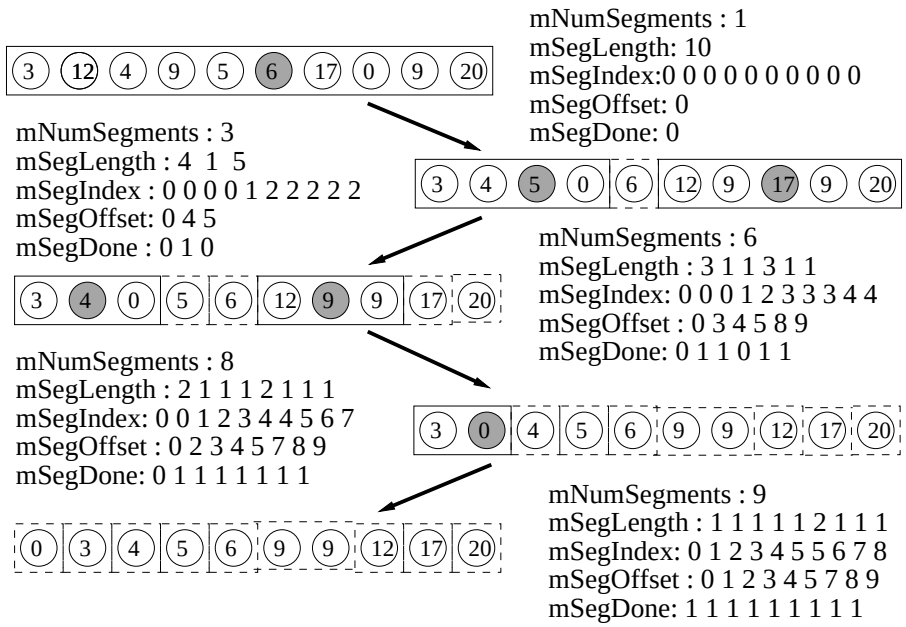


Figure 5.6: The modification to the segmented array for the quicksort. Shadowed elements are quicksort pivots. Elements in the same segment are grouped by rectangular boxes. Dotted boxes indicate segments that are not subject to recursive calls.

mNumSegments: size one. It stores the number of segments in a segmented array.

mSegIndex: array of size N. mSegIndex[i] returns which segment the i-th element belongs to.

mSegOffset: array of size mNumSegments. It stores the offset of each segment relative to the starting address of the data array.

mSegLength: array of size mNumSegments. mSegLength[i] returns the length of the i-th segment.

mSegDone: array of size mNumSegments. It is used for recursive calls. A "1" at position i means that the i-th segment has reached the exit condition of the recursive call.

In the quicksort example, such segment information also preserves the current state of the quicksort recursion. Figure 5.6 illustrates the status of a segmented array for quicksort. The six auxiliary arrays (32 bits each) come with a linear increase in the memory footprint for a total of 24 bytes per NESL data structure.

The layout of a segmented array can be dynamically modified by the user via storing a 1 in the mSegment array. The runtime is responsible for adjusting the remaining auxiliary arrays accordingly. We have developed an efficient data-parallel approach via Algorithm (Figure 5.7) to minimize the execution time of this operation. These auxiliary data structures together with the algorithm help reduce the overhead of the code generated by CuNesl.

In this parallel algorithm, all for loops and the Inclusive_Scan() function can be efficiently and

```
mSegIndex := Inclusive_Scan(mSegments, N);
Barrier();
mNumSegments := mSegIndex[N-1];
Barrier();
parfor i = 1 → N do
    mSegIndex[i] := mSegIndex[i] - 1;
end parfor
Barrier();
parfor i = 1 → N do
    if mSegments[i] == 1 then
        mSegOffset[mSegIndex[i]] := i;
    end if
end parfor
Barrier();
parfor i = 1 → mNumSegments do
    if i == (mNumSegments - 1) then
        nextOffset := N;
    else
        nextOffset := mSegOffset[i+1];
    end if
    mSegLength[i] = nextOffset - mSegOffset[i];
end parfor
```

Figure 5.7: Update auxiliary arrays from mSegments

cooperatively (independently) executed by SIMT threads. We need to generate two versions of code based on this algorithm to fulfill the need for different execution modes discussed in Section 5.4.2, one for the *kernel mode*, the other for the other two modes. In the *kernel mode*, all for loops are transformed into separate CUDA kernels and Inclusive_Scan() is invoked by calling appropriate CUDPP library APIs [67]. A Barrier() is implicitly enforced by the CUDA runtime. In the *block mode* and the *shared memory block mode*, the entire algorithm becomes a device function called by other device or global functions.This also applies to the Inclusive_Scan() function, which only needs to perform a local scan at the block level. Barrier() needs to be instantiated by __syncthreads() (provided as a CUDA device function) to ensure correctness.

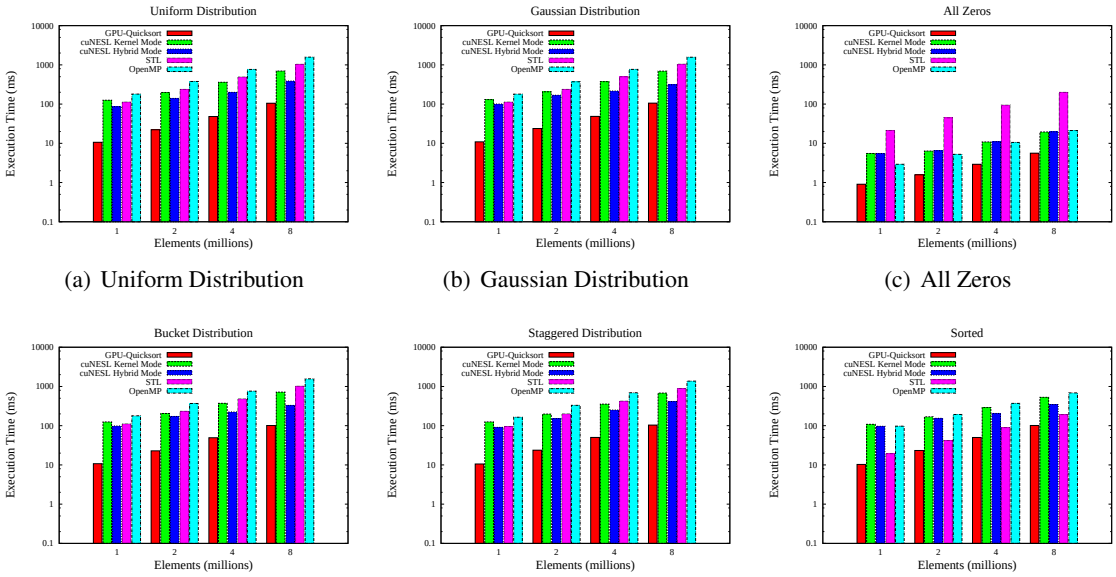
This strategy to provide kernel-level and block-level support for an operation needs to be applied to either pre-defined NESL primitives in the runtime or emitted code by the CuNesl compiler. This allows us to exhaustively explore different combinations of execution modes to find the fastest combination. Fortunately, except for a few differences (*e.g.*, barriers in *block mode* are realized via __syncthreads()), these two versions are similar to each other.

The implementation of the CuNesl runtime takes advantages of existing hand-crafted CUDA li-

braries for many of the parallel primitives supported in NESL. For example, CUDPP's APIs at different layers are heavily used in our runtime system. We also provide implementations of other primitives, such as sum, concatenation and reverse.

5.5.2 Optimizations

We call a segmented array a *regular* segmented array when all its segments are of the same length. For such segmented arrays, we do not need to waste memory and time to maintain the aforementioned auxiliary arrays. Instead, only a single scalar is needed to keep track of the length of each segment in the array. All other information, such as segment offset and the corresponding segment id for an element, can be calculated on-the-fly and independently by SIMT threads. The runtime system will convert a *regular* segmented array to a non-regular one whenever necessary.



(a) Uniform Distribution (b) Gaussian Distribution (c) All Zeros

(d) Bucket Distribution (e) Staggered Distribution (f) Sorted

Figure 5.8: Quicksort Results

5.6 Experimental Results

We conducted our experiments on a Quad-core Intel(R) Xeon(R) CPU E5507 machine with 6 GB memory. The GPU was a Geforce GTX 480 consisting of 15 Streaming Multiprocessors. The host code was compiled by Gcc 4.4.4. CUDA code was compiled by NVCC, CUDA release 4.0. Both Gcc and CUDA

```
1  function bitonic_sort(a) =
2  if (#a == 1) then a
3  else
4       let
5       bot = subseq(a,0,#a/2);
6       top = subseq(a,#a/2,#a);
7       mins = {min(bot,top):bot;top};
8       maxs = {max(bot,top):bot;top};
9       in flatten({bitonic_sort(x) : x in [mins,maxs]}) $
10
11 function batcher_sort(a) =
12 if (#a == 1) then a
13 else
14      let b = {batcher_sort(x) : x in bottop(a)};
15      in bitonic_sort(b[0]++reverse(b[1]))
16      $
```

(a) Batcher Sort in NESL

```
1  void batchersort(SegmentArray<T> &array) {
2    while (!array.isRecursiveAllDone()) { // first while loop
3      array.setRecDoneByLength(1);
4      // nothing to do, just push the segment info
5      array.pushSegments(0); // line 14 in NESL
6      array.bottop();
7    }
8    array.popSegments(0); // no action for the finest granularity
9    while (array.popSegments(0)) {
10     SubSegmentArray<T> bs(&array, Sub_Bot);
11     bs.reverse(); // correspond to the reverse call in line 15
12     while (!array.isRecursiveAllDone()) {// seond while
13       array.setRecDoneByLength(1);
14       if (array.isRecursiveAllDone()) break;
15       genMinMax(array); // responsible for line 5 to 9
16       array.bottop(); // deduced from subseqs in line 5 and 6
17     }
18   }
19 }
```

(b) Generated CUDA C++ code for Kernel Mode

Figure 5.9: Batcher Sort

codes are compiled at optimization level -O3.

5.6.1 Quicksort

We present CuNesl's quicksort performance by comparing with three other implementations:

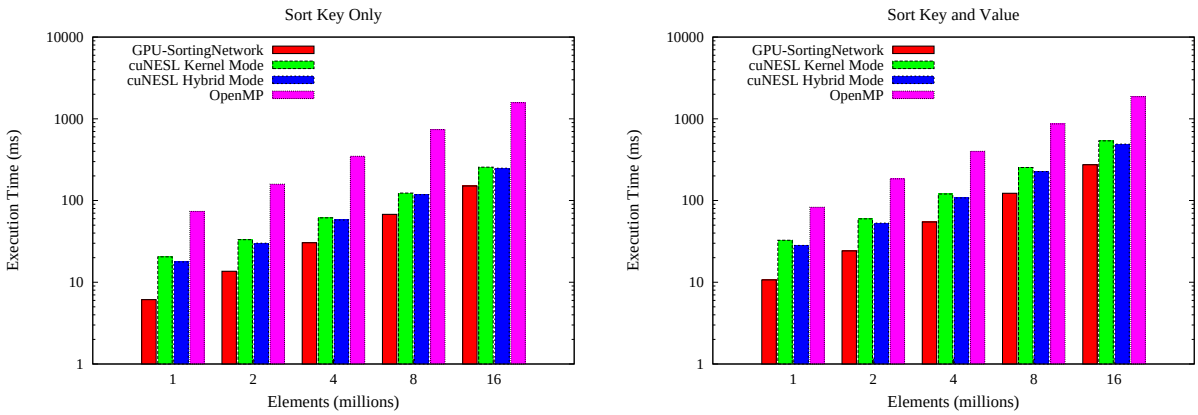
GPU-Quicksort: This is a hand-written CUDA sorting library using quicksort in the beginning and switching to bitonic sort in the end [27]. To the best of our knowledge, it is the fastest open-source GPU implementation involving quicksort. The total number of source code lines, including both the host-side C++ and CUDA, adds up to about 900 lines.

STL: This is also a hybrid sorting implementation: it first uses introsort, which is based on quicksort, followed by insertion sort. It is run on CPUs only.

OpenMP: We also wrote quicksort in OpenMP using the *parallel* pragma directives. This, too, is run on CPUs only. The maximal number of threads is 8. The same thread configuration applies to other experiments.

We use the number of lines of code (LOC) as a metric to assess the programmability, i.e., reflecting the effort of the programmer to write code.

For STL constructs, we are counting the lines of code at the first major level, e.g., inside of std::sort(). In our OpenMP implementation, we use std::partition() to split arrays into halves, which is a central part of quicksort. This hand-written STL code is counted as just one LOC in the table. The LOC metric shows that NESL supports extremely concise expressions of such a recursive function: The LOC metric



(a) Bitonic Sort (Key Only)

(b) Bitonic Sort (Key and Value)

Figure 5.10: Batcher Sort Results

is one to two orders of magnitude less than for the other implementations.

We adopted the same testing strategy as in [27] by measuring the execution time under different input distributions, namely uniform, Gaussian, zero, bucket, staggered and sorted. The details of these distributions are explained in [27]. We slightly revised the original Quicksort NESL script to choose a better pivot element for each segment array. Instead of blindly taking the element at the middle index, we pick the pivot as the average of the max and min value in each segment.

The final performance is shown in Figure 5.8. The Y axis shows the execution time on a log scale. The X depicts shows the array size from one million to eight million elements (numbers). For CuNesl, we show two bars. The first is obtained by only generating code in the *kernel mode*. The second starts with *kernel mode* and then switches to *block mode* after producing enough segments (256). This is referred to as the "hybrid mode" in the figures. The switching point needs to be tuned (currently manually, could be automated) because it depends on the size of the sorting data types and the resource usage (register and Shared Memory). We can see that the hybrid mode usually takes about half of the time of the *kernel mode*. This demonstrates our previous hypothesis that different execution modes are

Table 5.1: Quicksort: Line of Code Comparison

Implementation	LOC
GPU-Quicksort	900
CuNesl	9
STL	100
OpenMP	130

suitable for different segment arrays. We also tried to add the *shared memory block mode* to the hybrid mode when segments are becoming small enough fit in the GPU's Shared Memory. But it provides no improvement over the two-stage hybrid mode. The extra barrier and bookkeeping between the *block mode* and *shared memory block mode* resulted in a net performance loss due to overheads. Therefore, the execution time in this case is not displayed in the Figure 5.8.

Figure 5.8 shows that our best compiled quicksort routine (hybrid mode) is about two to three times slower than the hand-written CUDA implementation (GPU-Quicksort). This is mainly due to three reasons:

- GPU-Quicksort uses bitonic sort at the end, i.e., after spawning a sequence of the quicksort recursions. Quicksort is well known to be less efficient than bitonic sort due to the partition imbalance problem.

- GPU-Quicksort is using problem-specific knowledge to reduce execution time. For this particular case, the programmer knows that the concatenated total length from the less, equal and greater arras (partitions) are the same as the original array. This greatly increases the parallelization opportunity because the new offset for each element can be calculated independently inside a quicksort partition. Such information is difficult to deduce for the CuNesl compiler. Therefore, for safety reasons, a global scan needs to be performed to calculate the new offset in *"kernel mode"*. This enforces a barrier between different depths of recursion.

- For handwritten quicksort, programmers do not need to maintain auxiliary arrays for segmented arrays. They just need to keep record the sizes of each sub-array. (All other variants require these sub-arrays to support segmented arrays and incur overhead for maintaining these auxiliary data structures.)

The performance in all our cases is two to three times higher than STL, which is usually one third faster than our handwritten OpenMP implementation, except for the all-zero case. Considering the tremendous advantage in terms of programming effort, we believe that CuNesl is a viable way to realize data-parallelism for SIMT architecture.

5.6.2 Batcher Sort (Bitonic Sort)

We also evaluated the Batcher Sort benchmark, which recursively calls Bitonic sort in a depth-first manner. Bitonic sort itself is also a recursive call that keeps sorting symmetrical partitions in the first and second halves at different granularity levels. The NESL source code depicted in Figure 5.9(a) is almost as concise as that of quicksort.

This benchmark represents a typical example of multiple recursions. Correspondingly, CuNesl generates one while loop for each recursion. The top-level C++ code for the *kernel mode* is shown in

Table 5.2: Batcher Sort: Line of Code Comparison

Batcher Sort	LOC
SortingNetworks	250
CuNesl	15
OpenMP	120

Figure 5.9(b). Because the first-level recursion is operating in a bottom-up manner, we need to push the segment information onto a stack and invoke the second-level recursive functions (bitonic sort) when segments are popped (see the while loop at line 9). The second-level recursion is transformed into the while loop the same way as for the quicksort routine (see lines 12-17).

We compare CuNesl with two other implementations of Batcher Sort:

GPU-SortingNetworks: This code is released as an example in NVIDIA's CUDA SDK. It features highly optimized hand-crafted CUDA code.

OpenMP: We also rewrote Batcher Sort in C++ utilizing the OpenMP *parallel for* pragma directive for parallelization. This version runs on CPUs only.

The LOC summary is listed below. Again, CuNesl (NESL) increases the programmer's productivity as an order of magnitude fewer LOCs are required.

We applied batcher sort on two kinds of arrays: one is just a key array with unsigned int type; the other is a (key, value) pair array with unsigned int type for both key and value. Observed execution times are shown in Figure 5.10. Similar to quicksort, we provide two bars for CuNesl. One is obtained by executing in *kernel mode* only. The "hybrid mode" in this case means *kernel mode* followed by *shared memory block mode*. As shown in the figure, the "hybrid mode" is about 10% faster than the *kernel mode*. Given the fact that the *shared memory block mode* saves global memory traffic, it indicates that batcher sort is memory bandwidth bound. The same conclusion can be drawn for the other two implementations as well because they both take roughly twice as much time to sort the (key, value) pair as just the key array.

A closer look at the source code of GPU-SortingNetworks reveals that this program also divides the execution into two phases, where in the later stage it puts small sub-arrays into Shared Memory to reduce bandwidth consumption. This is exactly what CuNesl does. The handwritten CUDA code does not need to keep track of changes in the segmented array, making it about $30 - 40\%$ faster than the best CuNesl code (hybrid mode).

Batcher sort is more friendly to parallelization than quicksort, even though it only works for arrays of certain sizes (power of two). Within the investigated input size range (one million to eight million elements), batcher sort is twice as fast as quicksort on C++ code. CuNesl achieves up to a 5X speedup over the parallel OpenMP implementation.

5.6.3 Discussions

NESL's conciseness comes along with sacrifices: it can only pass limited information to the compiler. A human programmer can exploit algorithm-specific knowledge that a compiler cannot easily deduce. Therefore, we do not expect CuNesl's performance to be at par with hand-optimized GPU code. After all, it is often argued that the performance of a language is proportional to the required programming effort, especially for GPUs. Our results in the above two sorting algorithms show that the performance gap is not as large as the programming effort saved. The results are even more compelling when comparing CuNesl with codes running on CPUs. Our compiler outperforms them in terms of both execution time and programmability. In addition, there is still much room for CuNesl to improve its performance. Adding directives (e.g., OpenMP pragmas) maybe a promising direction for future research.

5.7 Future Work

Current development of CuNesl exposes many exciting directions we would like to pursue to make it more robust and efficient. Some are mentioned in previous sections. Additional ideas are listed below:

Auto-Tuning: At the current stage, the transition threshold between different execution modes is emitted as heuristic constants. Our reported result is obtained by manually tuning those constants. Our experience shows that changing those constants can sometimes make a significant difference in performance. It is thus desirable to auto-tune these parameters.

Non-Recursive Functions: This paper mainly focuses on how to transform recursive functions in NESL and optimize them. For non-recursive functions, we would like to show that CuNesl performs equally well by transforming independent code schemes into segments.

Scheduling of Execution Mode: Right now, the switching between different execution modes is hand-coded: a barrier exists that prevents two execution modes from overlapping in time. By aggressively scheduling modes in parallel, we may be able to obtain better performance for irregular algorithms, such as quicksort.

5.8 Conclusions

This chapter presents translation techniques for a nested data parallel language to be efficiently executed on modern SIMT architectures. Previous approaches to convert nested parallelism into flattened segments failed to consider the hierarchy of execution modes of modern architectures. We show that by applying control-flow transformations on the flattened code, the new recursion-free control flow provides the freedom to dynamically transition between different threading models. The resulting CUDA code allows the user to enjoy both the conciseness of data-parallel languages and the computational power of SIMT accelerators.

Chapter 6

HiDP: A Hierarchical Data Parallel Language

6.1 Introduction

Contemporary research seems to indicate that no panacea can seamlessly adapt sequential legacy programs to modern parallel architectures. Simply converting programs into many concurrently executing threads may not necessarily deliver expected levels of performance improvements. This is partly due to today's parallel machines consisting of far more complicated execution and memory hierarchies than the simplistic Von Neumann model, which may be a suitable abstraction for sequential programs — but not so much for today's hierarchical parallelism. Any approaches that ignore such hierarchies are likely to yield performance inferior to they capabilities.

Consider modern GPU architectures as an example. Table 6.1 lists the execution hierarchies in Nvidia GPUs. For a single GPU unit, there exist at least four execution levels, each of which features different favorable degrees of parallelism and synchronization methods. *Suppose a problem can be divided by a number of concurrent tasks, which can be realized by fine-grained data-parallel threads cooperatively.* When the number of tasks is small (a few to a few dozens), the top *kernel* level suffices to utilize all GPU computing resources. This is done by assigning multiple blocks to one logic task. But this requires the local barrier synchronization in one task to be replaced by a more expensive global barrier because of a lack of hardware synchronization across multiple blocks. We can also assign just one block to process one task. The benefit of doing this is that task barriers can be implemented locally inside each block. But this is only beneficial when the number of tasks is large enough to exercise all GPU resources. As we delve down to the warp or thread level, synchronization comes at no cost because it is ensured by SIMD and instruction ordering. However, GPUs need substantially more parallelism to reach their peak instruction throughput. There is no clear delimiter to the range of suitable parallelism for each execution level. Effective hierarchical parallelization hence often become dependent on both

the hardware and the application. This ambiguous boundary exposes more challenges to languages, as it becomes the task of the compiler and runtime to decouple hierarchical parallelism from the algorithmic expressions.

Hierarchical architectures like GPUs have a substantial influence on the way to solve problems. A common approach to match the hardware structure involves a two-step decomposition. Firstly, a task-level divide-and-conquer approach partitions a large-scale problem into a number of smaller tasks. This step provides opportunities for reducing synchronization overhead and utilizing faster but limited on-chip caches. Secondly, these tasks are executed by a massive number of threads cooperatively, demanding exposition of fine-grained data parallelism. As a result, data-parallel primitives, such as segmented parallel scan and reduction (detailed later), play an important role in coding productivity and performance.

Fortunately, many seemingly inherently sequential operations (reduce, scan, partition, etc.) have efficient parallel solutions. A significant effort has been invested by experienced programmers in providing such solutions, often as libraries. In the CUDA ecosystem, libraries like Thrust [65] and CUDPP [67] are widely used by developers to avoid implementing such operations from scratch. However, users often find it difficult to integrate these libraries with their own code for mainly two reasons: (1) User code and libraries are often required to be closely coupled. Users often have to acquire fine details of the library. (2) Libraries often provide alternate implementation choices of the same functionality due to the hierarchical execution model in today's microprocessors. The best choice is often data dependent, yet not necessarily obvious to programmers. Therefore, it is necessary to try each of them, which can be a daunting task for end users.

Contributions: We propose HiDP, a data-parallel language with hierarchical parallel-for clauses and built-in data-parallel primitives. These language features are equally suitable for describing parallel algorithms and for obtaining high performance in contemporary GPUs. The HiDP compiler judiciously maps parallel for constructs onto the hierarchical execution model of modern GPUs. When multiple mapping choices exist, the compiler generates different versions of code, one for each mapping. It also emits tuning code that aids users in selecting the appropriate version, or the user can manually prune alternatives based on domain knowledge.

HiDP is a machine-independent language. In fact, HiDP encourages users to express algorithms in a general, architecture-neutral fashion. This makes HiDP robust to future architectural advances and extensible for code generation in other formats, such as OpenCL [69]. Like many other high-level languages, HiDP is very concise and easy to learn. More than an order of magnitude of lines of code can be saved compared to native CUDA code. HiDP could also serve as an intermediate language for other high-level languages since its code transformations result in performance that only marginally falls short of hand-written CUDA code.

6.1.1 A Simple Motivational Example

As a motivation to generate multiple kernels, consider a simple example with several implementation alternatives to calculate a uniform segmented reduction on a large 32-bit integer array. We fix the size of the array at 16 million numbers and vary the segment size from 1 to 128K. Four approaches are implemented: 1) Reduce each segment by a block; 2) reduce each segment by a warp (32 threads); 3) reduce each segment by a partial warp (8 threads); 4) reduce each segment by a single thread. We keep the block size constant at 256 threads in all approaches. The measured resulting GFlops are shown in Figure 6.1. As we can see, none of the implementations outperforms the others all the time. The thread approach works best when the segment size is very small because other methods dedicate too many resources to a single segment. As the segment size increases, the partial warp catches up because it exhibits a more efficient memory access pattern. The warp approach delivers the best GFlops starting at segment size of 128.

This is an over-simplified case. Reduction in real-world applications is usually mixed with other computations in the same kernel, and the segment sizes are not necessarily uniform. This makes it even harder to decide which (single) implementation works best.

The rest of the chapter is organized as follows. The HiDP language is introduced in Section 6.2. A step-by-step description of the compiler and run-time framework is discussed in Section 6.3. Experimental results are illustrated in Section 6.4 to exemplify the efficiency of our proposed system. This is followed by related work in Section 6.5. Future work and a summary are provided in Section 6.6 and 6.7, respectively.

6.2 The HiDP Language

A HiDP program is built around a top-level structure specified through the keyword *function* followed by the function name. Other functions can be defined and can be called by the top-level function or by each other. Yet, there is only a single entry to a HiDP program. The compiler will generate a legal C++ function signature and body based on the top-level function.

The header of the function body declares the arguments of this function. The data flow and read-

Table 6.1: Execution Hierarchies in Modern GPUs

Execution Level	Suitable Parallelism	Synchronization
Kernel	less than a few dozens tasks	kernel boundaries
Block	a few dozens to a few hundreds	$\_\_syncthreads()$
Warp	more than a few hundreds	Not Necessary
Thread	more than tens of thousands	Not Necessary

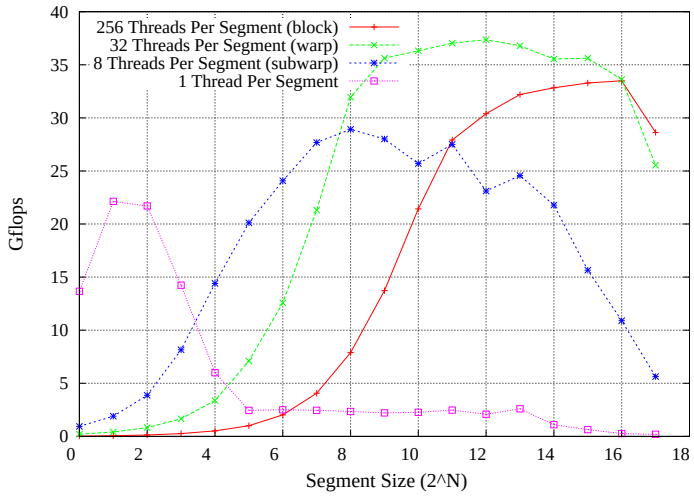


Figure 6.1: A Simple Example Showing Performance Sensitive to Execution Model

/write access properties inside the function are indicated by keywords *input*, *output* or *inout*. All arguments of a function are passed by reference, *i.e.*, a change of an argument in the function will be seen by the function's callee.

In designing the HiDP language, we pursue the following major goals: We intend to

- expose low-level data structures for full control over the data layout design;
- preserve the conciseness of data parallel script languages;
- provide the ability to customize data-parallel operations, *e.g.*, hierarchical or partial mappings;
- embed basic data-parallel primitives in the language to improve coding productivity;
- keep the language platform independent and only add machine-dependent information at the directive level.

In the following subsections, we present key aspects of HiDP and explain how our goals are met by them.

6.2.1 Data Types

HiDP's basic structure is an array of any dimension greater or equal to 0. (Scalars have a dimension of 0.) There are two ways to declare a variable. One is at the function header (argument), the other is inside the function body (local variable):

data_type var([dim0]...[dimn]);

A declaration starts with a data type identifier, which can be either a fundamental C/C++ data type (char, float int ...) or a derived (user-defined) one. The number of bracket pairs after the variable name implies the dimension of the variable. The size in each dimension of these arguments is expressed symbolically in terms of either constant or free variables, the values of which must be determined by the HiDP runtime system. An example of declaring a 2D dimensional float array and a 1D dimensional float array is as follows:

float my_2d_array[I][J], my_1d_array[K];

The other option for declaration is to specify a data type for a scalar integer variable at the beginning of a *map block* (see Section 6.2.3). Such a scalar integer has to be within a certain range, which is expressed as two tuples enclosed by brackets (inclusive) or parentheses (exclusive). We call this kind of variable a *map iterator*:

map_iter := ranges;

The range is relaxed to be any arithmetic expression of variables and constants. An example of declaring a map iterator i from 1 to $J-1$ is:

i := [1 : J−1);

We will discuss the *map iterator* in more detail in Section 6.2.3.

6.2.2 Data Parallel Expressions

Like many other data-parallel languages, HiDP allows concise array operations on each element of a structure. This corresponds to the concept of an *apply-to-each* or *map* construct in other languages. Such expressions, together with the *map block*, form the fundamental statements of the HiDP language. Consider the statement

A = B ∗ C;

All elements of A are updated by the multiplication of elements at the same relative position in B and C. HiDP requires that all variables in a data parallel expression maintain the same shape (same number of dimensions and same size on each dimension) but allows scalar variables to "expand" to the same shape as other multi-dimensional variables in the same expression.

6.2.3 Hierarchical Map Blocks

The support for data parallel expressions above improves coding productivity by eliminating some *for* loops of languages like C and C++. But the default *apply-to-each* behavior may be too strict to express certain algorithms.

HiDP relaxes its stringent behavior by defining a *map block* following the principle idea of a *parallel for* construct. The number of iterations inside a *map block* is determined by *map iterators*, which must

be defined at the beginning of the *map block*, but there may be multiple ones of them defined for each *map block*. In addition, HiDP allows an optional suffix function call to be made at the end of the *map block*. Therefore, a *map block* can be of the following formats (following EBNF notation):

```
map_block: map block
         | map block suffix
block: { statements }
suffix: function_call
```

Map blocks can be hierarchical. A *map block* is called another *map block*'s parent if the former fully encloses the later. Two types of statements can reside in a *map block*: scalar expressions and pre-defined data-parallel primitives (see Section 6.2.4). The *parallelism* of a *map block* is determined by the product of all *map iterators* of itself and all its parent *map blocks*. The level of *parallelism* is expressed by the number of concurrent scalar expressions executed in this *map block*. HiDP assumes sequential execution of instructions in the *map block* but does not assume any synchronization between different iterators (except for entering and exiting data parallel primitives, see Section 6.2.4). Therefore, the behavior of any writing to the same memory position from different iterators is undefined.

Many applications have inherent nested parallelism. This fits naturally with HiDP's hierarchical map blocks. Starting from the outermost map block, the compiler's major role is to determine which execution model is best suited for this level, optionally enhanced by programmer hints.

6.2.4 Data Parallel Primitives

HiDP supports many data-parallel primitives that improve coding productivity. Those primitives can be written either outside any *map block* or inside/appended to a *map block*. If associated with a map block, primitives implicitly call local barriers before entering and after exiting the block. Primitives inside the map block can be regarded as segmented primitives. All operations are performed independently within each segment. Each segment may execute within several blocks cooperatively, within a single block, within a warp or even within a thread. This depends on the number of segments and availability of the primitive's implementation at the execution level. Such choices are ultimately made by the HiDP compiler and runtime. HiDP requires each irregular segmented array to be associated with two index vectors, termed low_range and $high\_range$. These vectors indicate the low and high indexes of each segment, respectively. This representation of a segmented array differs from NESL [22], where an associated boolean array of the same size as the original array is used to infer segment boundaries. For regular segmented arrays (segment sizes are the same), HiDP supports a different interface where only two scalar inputs are used to replace the two low and high index vectors: seg_size and num_seg. This design choice was driven by practical considerations. We find that significant performance benefits can sometimes be achieved if prior knowledge about regularity is available. Table 6.2 shows the syntax of selected data-parallel primitives in different scenarios. For example, a typical usage of a partition in a map clause is:

```
float in[size], out[size], pivots[num_segs];
int low[num_segs], high[num_segs], out_low[num_segs * 2],
new_high[num_segs *2];
...
map {
  seg := [0: num_segs);
  partition(in, out, low, pivots, low, high, new_low, new_high, MyCompare);
}
```

There will be num_segs instances of partition operations on the input array in. The ith instance ($0 <= i < num\_segs$) works on elements between $low[i]$ and $high[i]$. The new index ranges for the two new smaller partitions are created in $new\_low[2 * i], new\_high[2 * i], new\_low[2 * i + 1]$ and $new\_high[2 * i + 1]$. $MyCompare$ can either be a native CUDA device function or an internal HiDP function.

Depending on the position of the data parallel primitive, there may be multiple instances of primitive calls. For example, if a data primitive is called inside a *map block*, the number of instances is the *parallelism degree* of the current *map block*. These instances can be executed in parallel without any synchronization. But HiDP assumes local barriers before and after each of them. In other words, the range of the synchronization is constrained to the necessary range to guarantee correctness of each instance. If a primitive is a suffix function call for a *map block*, only a single local barrier constrained by the *ranges* is needed.

6.2.5 User-Assisted Directives

HiDP supports directives as annotations for map clauses. They are required to assist the compiler in performing the mapping from a hierarchical structure to an execution model. They often require prior knowledge that cannot be deduced by the compiler, and they help reduce the exploration space. Such directives must immediately precede the *map clause* in the program. Their syntax is:

```
#pragma hidp [kernel|block|warp|subwarp|thread]
```

6.2.6 GEMM in HiDP

As a concrete example, consider the HiDP source code for the level-3 BLAS GEMM routine in Figure 6.2. Lines 2 to 4 define the function header. The body of the function consists of a single-level map structure with a reduce suffix on temporary variable $\_c0$. Line 8 defines three *map iterators* for the *map block*. The reduction is applied to all k iterators (line 10) for different i and j iterators and is assigned to the 2-D array $C1$. As mentioned above, synchronization is implicitly reinforced before and after the suffix reduction call, but only at a local range (for every k iterators). After $C1$ is updated, C is finally calculated by the GEMM rule (line 11).

Table 6.2: Selected Data Parallel Primitives in HiDP

Irregular Parallel Primitives
$\_min/\_max = min/max(input, low\_range, high\_range)$
$sort(in\_key, out\_key, [in\_value, out\_value], low\_range, high\_range, dir)$
$partition(in, out, [in\_value, out\_value], in\_low\_range, pivots,$ $in\_high\_range, out\_low\_range, out\_high\_range, function)$
$reduce(\text{"}+/*\text{"}, input, output, low\_range, high\_range)$
$scan(in, out, low\_range, high\_range)$
$reverse\_inplace(inout, low\_range, high\_range)$
$reverse(in, out, low\_range, high\_range)$
Regular Parallel Primitives
$\_min/\_max = reg\_min/\_max(in, out, seg\_size, num\_seg)$
$reg\_sort(in\_key, out\_key, [in\_value, out\_value], seg\_size, num\_seg, dir)$
$reg\_reduce(\text{"}+/*\text{"}, in, out, seg\_size, num\_seg)$
$reg\_scan(input, output, seg\_size, num\_seg)$
$reg\_reverse(inout, seg\_size, num\_seg)$
$reg\_reverse(in, out, seg\_size, num\_seg)$
Outside Map Block
$\_min/\_max = min/max(in, size)$
$sort(in, out, size, dir)$
$partition(in\_key, out\_key, [in\_value, out\_value], pivot, size, function)$
$reduce(\text{"}+/*\text{"}, in, out, size)$
$scan(in, out, size)$
$reverse\_inplace(inout, size)$
$reverse(in, out, size)$
Map Suffix Functions
$reduce(\text{"}+/*/min/max\text{"}, output, input, ranges)$

```
1  # implementing C = alpha ∗ A ∗ B + beta ∗ C
2  function GEMM
3  input float _alpha, _beta, A[M][K], B[K][N];
4  inout float C[M][N];
5  {
6    float C1[M][N];
7    map{
8      m:=[0:M); n:=[0:N); k:= [0:K);
9      _c0 = a[m][k] ∗ b[k][n];
10   } reduce("+", C1[m][n], _c0, k:=[∗]);
11   C = _alpha ∗ C1 + _beta ∗ C;
12 }
```

Figure 6.2: GEMM in HiDP

HiDP encourages users to express algorithms at the finest data granularity. For the GEMM example, this occurs at line 9 in Figure 6.2, where the element-wise multiplication over all three dimensions is expressed. This makes HiDP independent of the underling hardware architecture. The decisions on whether or not to fuse them and at which level are left to the compiler backend as it depends on the properties of the targeted hardware.

Table 6.3: 1-D Shapes of Execution Model

Execution Level	s_kernel	s_block	s_warp	s_sub-warp	s_sub-warp2	s_thread
1-D Shape	gridDim.x/BLOCK_PER_TASK	gridDim.x	s_block * WARP_PER_BLOCK	s_warp * 4	s_warp * 8	s_block * blockDim.x

Table 6.4: 1-D Shapes of Execution Model Given its Immediate Upper Layer

Level	kernel	block	warp	sub-warp	sub-warp2	thread
s_kernel	1	-	-	-	-	-
s_block	BLOCK_PER_TASK	1	-	-	-	-
s_warp	s_block ∗ WARP_PER_BLOCK	WARP_PER_BLOCK	1	-	-	-
s_sub-warp	s_warp ∗ 4	s_warp ∗ 4	4	1	-	-
s_sub-warp2	s_warp ∗ 8	s_warp ∗ 8	8	4	1	-
s_thread	s_block ∗ blockDim.x	blockDim.x	32	8	4	1

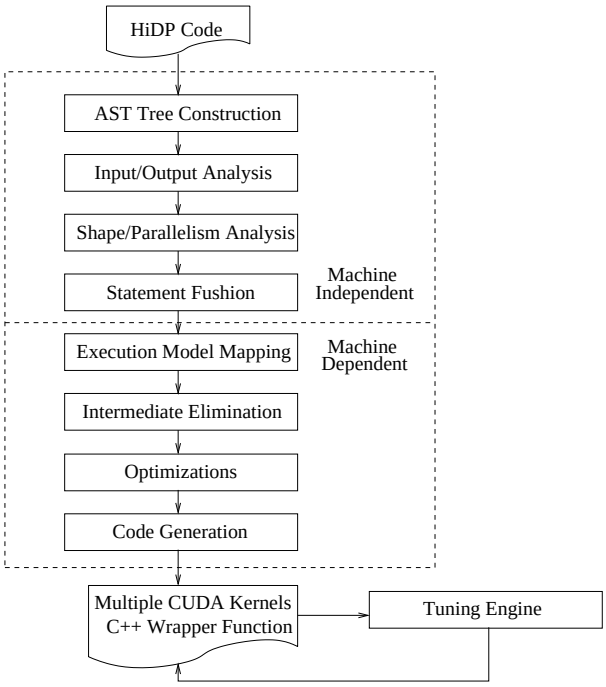


Figure 6.3: Overview of HiDP Compiler

6.3 The HiDP Compiler

In this section, we provide an overview of the compilation steps to transform HiDP into a set of CUDA/C++ functions containing both the host and device code. We will use the GEMM example in the previous section as a running example.

6.3.1 Overview

The HiDP compiler consists of a number of phases shown in Figure 6.3. The input of the framework is a HiDP program with a single function entry. It emits one or even multiple versions of the CUDA kernel code and C++ host code for the same HiDP program. If the output contains multiple versions, a wrapping C++ function is also generated to aid during the tuning process. Users can intercept the tuning process and directly pick the most appropriate version.

6.3.2 Front End

The HiDP compiler parses each routine to transform a HiDP program into an abstract syntax tree (AST). It detects the top entry-level function and instantiates other internal functions at the top level. There are four types of statements in HiDP: assignment expression, map block, branch block and function call.

They are hierarchical in the sense that a map block can contain multiple assignment expressions inside. All further analysis is performed hierarchically at each statement level. The HiDP front end does not expand data-parallel expressions into for loops throughout the code transformations.

After parsing, the compiler recursively analyzes the input and output set of each statement from the bottom up. Having a complete understanding of this step is important for generating function arguments and recovering inout data in the tuning wrapper.

6.3.3 Nested Shape Representation and Analysis

HiDP relies heavily on shape analysis to perform statement fusion and execution model mapping in a safe manner. To achieve that, each statement is analyzed to obtain its shape, which indicates the maximal possible number of data-parallel threads for this statement. We call it the *parallelism degree* of a statement. Parallelism degrees can be multi-level depending on the position of a statement. We use the notation of {[level 0], ..., [level n-1] } to represent an n-level shape. For the GEMM example below, there are four statements in the HiDP source code: a map block (s1), an assignment inside the map (s2), a suffix function call (s3) and another assignment outside any map block (s2). The shape analysis starts from the innermost assignment (s2). Its shape is determined by the range of all map iterators at the same or higher levels. In this case, there is only one map block. Therefore, its shape is a single-level 3D shape of $\{[0:M, 0:N, 0:K]\}$. Next, the reduce call is analyzed. The reduction range is for all ks. Therefore, $[0:K]$ is promoted to the next level making the reduce function's shape a two-level shape of $\{[0:M, 0:N], [0:K]\}$. This is a two-level shape where each instance in the first level shape space ($M \times N$) contains up to K degrees of data parallelism. Its instances need local barrier support (reduction in this case). The shape of the map block is kept consistent with its suffix function call. The shape of the last statement is deduced from the dimension of its operands ($C1$ or C). Hence, it is a single level shape ($\{[0:M], [0:N]\}$). The shape of each statement after shape analysis is shown below:

```
float C1[M][N];
map{ # { [0:M, 0:N], [0:K] } (s1)
    m:=[0:M); n:=[0:N); k:= [0:K);
    _c0 = a[m][k] * b[k][n]; # { [0:M, 0:N, 0:K] } (s2)
}reduce(''+'', c1[m][n], _c0, k:=[*]); # { [0:M, 0:N], [0:K] } (s3)
C = _alpha * C1 + _beta * C; # { [0:M, 0:N] } (s4)
```

6.3.4 Statement Fusion

The main motivation behind combining as many operations as possible into a kernel is to save off-chip memory transactions because intermediate variables can be kept in registers, which avoids accesses to global memory. The HiDP compiler tries to merge statements at the top level ((s1) and (s4) in the GEMM example) building on shape analysis. Two statements can be fused if and only if their shapes are

compatible with each other. Two shapes are compatible when one is a prefix of the other in a flattened format. In the GEMM example, (s4)'s shape is a prefix of (s1). Therefore, they can be fused into a larger unit. (s4)'s shape extends to the same number of level as (s2), while the parallelism degree in the second level is just 1. After fusion, the GEMM function becomes a single statement, as shown in the following:

```
{
  map{ # { [0:M, 0:N], [0:K] }
    ...
  }
  C = _alpha * C1 + _beta * C; # { [0:M, 0:N], 1 }
} # { [0:M, 0:N], [0:K] }
```

The fused statement does not necessarily correspond to a single kernel at this point. Its transformation also depends on which execution model it is mapped to. For example, if the compiler later decides to assign multiple blocks to execute one instance in the $M \times N$ space at the first level shape, a barrier is needed for the reduction. This results in multiple kernels due to the lack of a global barrier across multiple blocks in CUDA.

6.3.5 Execution Model Abstraction and Mapping

Starting with this phase, the transformations are machine dependent. First of all, we depict the target machine as a set of hierarchical execution models. HiDP currently only supports CUDA in the backend. We will thus use the CUDA terminology throughout the rest of the section (even though OpenCL or OpenMP mappings are feasible as well). We add two more execution models to the one mentioned in Section 6.1. We call them sub-warp (8 thread lanes) and sub-warp2 (4 thread lanes). Similar to statements in HiDP, each level has a physical shape, which corresponds to the number of parallel instances at this level in GPUs. Table 6.3 lists the one dimensional shapes for all supported execution models. The job of execution model mapping is to associate the hierarchical statement shape into appropriate physical shapes according to their *parallelism degrees*. The physical shape of an execution model also depends on its immediate upper layer during the mapping. The relative shape for each case is shown in Table 6.4. The lower level shapes always have equal or more parallelism than the upper level shapes.

Take GEMM as the example: The shape of the fused statement is $\{[0 : M, 0 : N], [0 : K]\}$. The first level has $M \times N$ parallelism degrees. Since these are inputs to the function and are not known at compile time, HiDP may select any of the execution models, assuming $M \times N$ ranges from one to arbitrarily large number. The switching point is marked as a tuning parameter. The second level K is always mapped to the thread level because it is the last level. To conserve space for depicting the code, we prune the tuning space from 6 possibilities to 3 by choosing only block, warp and thread for the first level shape mapping. In fact, this can also be done by inserting a pragma before the map block:

```
#pragma hidp block warp thread
```

The set of valid mappings are shown in the table below. The expressions in parentheses represent the physical parallelism degree at this level.

[0:M, 0:N]	[0:K]
block(gridDim)	thread(blockDim)
warp(gridDim * WARP_PER_BLOCK)	thread(32)
thread(gridDim * blockDim)	thread(1)

After this step, we are able to determine CUDA kernel delimiters for each mapping. Because our run-time supports in-kernel local barriers for the three mappings we choose, a single kernel can implement the fused statement. The scope of the local variable $C1$ is within the kernel and its access pattern is strictly sequential, meaning that each scalar in the array is accessed by the same iterator. Therefore, it can be kept in the register file without any writes to the global memory, obviating its storage allocation.

6.3.6 Machine Dependent Optimizations

An important optimization strategy for CUDA code generation is to take advantage of the fast on-chip Shared Memory. The HiDP compiler tries to detect shared access patterns between neighboring threads. Again, this depends on the final execution mapping. HiDP searches for arrays whose indices contain only constant or map iterators that are mapped to the thread execution model. HiDP reasons about the shape of thread layout to facilitate the loading of shared data.

6.3.7 Loop Unrolling and Code Generation

The final code generation step needs to consider the mismatch between the *parallelism degree* of the nested shape (usually data dependent) and the physical parallelism of the corresponding execution model. The former is often greater than the latter. Because the execution order of iterators in the same map block is irrelevant, we generate a for loop with the following template:

for (id = iter_start + level_id; id $<$ iter_end; id+=level_stepsize) {
    iterator = id;
    ... (loop body); }

where $iter\_start$ and $iter\_end$ are the left and right boundaries of the map iterator. Furthermore, $level_stepsize$s are the same as the values in Table 6.4 for the case of a one dimensional shape.

Each supported data-parallel primitive has properties like shared memory usage and auxiliary variables. The properties are carried through the compiler framework and are interlaced with other HiDP code. On the host side, all arrays are encapsulated by the HiArray class, which supports an arbitrary number of dimensions and maintains data integrity according to the read/write properties deduced by the

compiler. *It is not mandatory to support data-parallel primitives in every level of the execution model.* Any restriction is considered by the compiler to conduct the execution model mapping.

6.3.8 GEMM CUDA/C++ Output

```
1  // gemm block version
2  __global__ void gemm_block(...) {
3    __shared__ float sum[BLOCK_SIZE]; //
4    int m = blockIdx.x;
5    int n = blockIdx.y;
6    float s = 0.0f;
7    for (k = 0 + threadIdx.x; k < K; k += blockDim.x)
8      s += fetchA(m,k) * fetchB(k,n);
9    reduce_block<ADD, float>(sum, s);
10   if (threadIdx.x == 0)
11     fetchC(m,n) = alpha * sum[0] + beta * fetchC(m,n);
12 }
13 // gemm warp version
14 __global__ void gemm_warp(...){
15   __shared__ float sum[WARPS_PER_BLOCK][WARP_SIZE];
16   int warpId = threadIdx.x/WARP_SIZE;
17   int warpIndex = threadIdx.x & 0x1F;
18   int m = blockIdx.x;
19   int n = blockIdx.y * WARPS_PER_BLOCK + warpId;
20   float s = 0.0f;
21   for (k = 0 + warlIndex.x; k < K; k += WARP_SIZE)
22     s += fetchA(m,k) * fetchB(k,n);
23   reduce_warp<ADD, float>(&sum[warpId][0], s);
24   if (warpIndex == 0)
25     fetchC(m,n) = alpha * sum[warpId][0] + beta * fetchC(m,n);
26 }
27 // gemm thread version, before shared memory optimization
28 __global__ void gemm_thread(...) {
29   int m = blockIdx.x * blockDim.x + threadIdx.x;
30   int n = blockIdx.y * blockDim.y + threadIdx.y;
31   float s = 0.0f;
32   for (k = 0; k < K; k += 1)
33     s += fetchA(m,k) * fetchB(k,n);
34   fetchC(m,n) = alpha * s + beta * fetchC(m,n);
35 }
```

Figure 6.4: HiDP Emits Different Kernels

Figure 6.4 lists the emitted GEMM kernel code for the aforementioned three mappings. Depending on the actual execution model mapping, HiDP emits different reduce functions (lines 9 and 23 for block and warp versions, but none for the thread version). The assignment expression inside the map block in

HiDP code is converted into for loops (lines 7 to 8, 21 to 22, 32 to 33) according to the template mentioned in Section 6.3.7. Finally, special care needs to be taken when there is more physical parallelism than the shape parallelism at a certain level. The expression needs to be ensured to only enable the first few threads. This is the case for the (s4) statement because its shape is 1 on the second level, but there are multiple valid degrees of physical parallelism for the block and warp versions. Consequently, lines 10 and 24 are inserted by the compiler to adjust for the parallelism difference.

The host C++ code is generated with necessary branches to choose which version of the kernel to run (lines 11, 14 and 17 in Figure 6.5). The parameters TUNING_0 and TUNING_1 are tuning parameters that need to be determined later.

6.3.9 Auto-Tuning

If the compiler detects any tuning possibilities, it will also emit tunable code wrapped by profiling code to measure the execution time for each code path. The user can run the executable in the *tuning mode*, where the measured time for each training test case is reported. The second part of Figure 6.5 illustrates this concept. *After the training phase, the user can then launch an analysis tool* operating on the profiling results to determine appropriate switching conditions for different versions of generated code.

Our analysis tool performs a linear regression match to determine the best time to switch kernels. Occasionally, a switch may not be result in performance benefits because other factors besides the detected *parallelism degree* affect performance but are not factored into decisions. If that was the case, users could always overwrite HiDP's decision by supplying customized code around various kernels to select the best one based on their prior domain knowledge.

6.4 Experimental Results

In this section, we investigate the performance of HiDP's generated code in several examples. Not only do we compare with parallel implementations on CPUs in some cases, but also with published, hand-optimized CUDA implementations of the same workload. As we will see, even the compiler cannot apply some of the optimizing techniques that an experienced programmer can, while our auto-tuning scheme, an optimization phase that is usually ad-hoc or absent in hand-written code, closes this performance gap.

The experimental platform is a two-socket machine with two AMD Opteron 6128 processors (8 cores each), one Nvidia GTX 480 and 32 GB memory. All experiments are performed using single-precision floating point, unless stated otherwise.

```
1  // generated wrapper code with tuning branches
2  void gemm_wrapper(HiArray<float, 1> &C, ...){
3    int C_dim0 = C.getDim(0); int C_dim1 = C.getDim(1);
4    ...;
5    int M = C_dim0; int N = C_dim1;
6    ...
7    vector<int> degree_0;
8    degree_0.push_back(M);
9    degree_0.push_back(N);
10   dim3 block, grid;
11   if (degree_0 < TUNING_0) {
12       config_block(degree_0, block, grid);
13       gemm_block<<<...>>>(...);
14     } else if (degree_0 < TUNING_1) {
15       config_warp(degree_0, block, grid);
16       gemm_warp<<<...>>>(...);
17     } else {
18       config_thread(degree_0, block, grid);
19       gemm_thread<<<...>>>(...);
20     }
21 }
22 // tuning function
23 void gemm_tuning(HiArray<float, 1> &C, ...){
24   int C_dim0 = C.getDim(0); int C_dim1 = C.getDim(1);
25   ...;
26   int M = C_dim0; int N = C_dim1;
27   ...
28   vector<int> degree_0;
29   degree_0.push_back(M);
30   degree_0.push_back(N);
31   dim3 block, grid;
32   for (int i = 0; i < 3; i++) { // three paths
33         save_inout_arrays();
34         start_timing();
35         gemm_block<<<...>>>(...); // for i == 0
36         gemm_warp<<<....>>>(...); // for i == 1
37         gemm_thread<<<...>>>(...); // for i == 2
38         end_timing();
39         report_timing(degree_0); }
40 }
```

Figure 6.5: Generated C++ Code by HiDP Compiler (Code is expanded for clarification purpose. Actual code may differ)

6.4.1 GEMM

Following the GEMM example in previous sections, we compare with the GEMM of the CUBLAS 4.2 library, a hand-crafted BLAS implementation released by Nvidia.

Let the sizes of the three matrices in $C = alpha \times C + beta \times A \times B$ be $M \times N$ (for C), $M \times K$ (for A) and $K \times N$ (for B). As mentioned in previous sections, our compiler generates several versions

of CUDA code depending on the size of $M \times N$, *i.e.*, the parallelism degree detected by the compiler. The auto-tuning engine will find the best switching points after several iterations of off-line training.

Figure 6.6 depict the results for double precision matrix-matrix multiply where K is 4096 while varying $M \times N$ from 16 to 65536. The figure shows the absolute execution time for each case (except some long execution time for block and warp versions at $M \times N >= 4096$). As we can see, when $M \times N <= 64$, the block version outperforms other techniques. This is because assigning an entire block (at this size) to cooperatively compute one element in C has a better chance of saturating GPU resources than assigning one thread per element. As the parallelism ($M \times N$) increases, the warp version catches up and performs best in the range of $128 <= M \times N <= 256$. For $M \times N$ exceeding 256, HiDP's thread version outperforms the other two versions. In contrast, CUBLAS does not deliver the best performance until $M \times N$ reaches 32768. This implies that the hand-written CUBLAS assigns multiple elements per threads (*a.k.a.* thread fusion), which hurts performance for cases when $M \times N$ is small to medium.

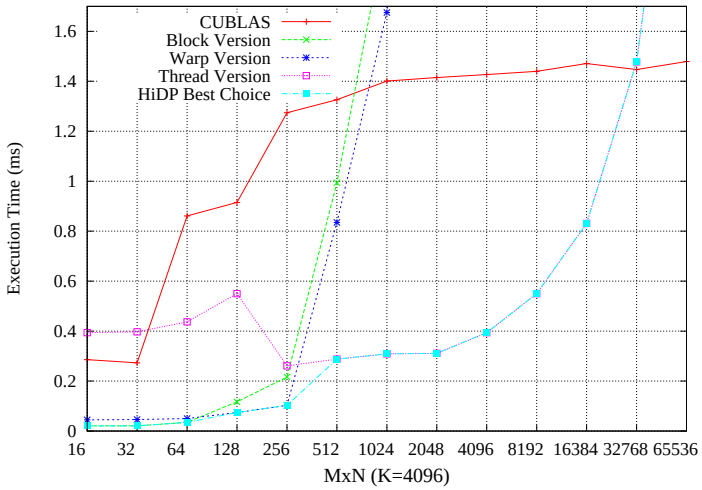


Figure 6.6: GEMM Execution Time for Small and Medium $M \times N$s

Of course, this is by no means to say HiDP can replace the GEMM in CUBLAS. For large-scale GEMM, CUBLAS outperforms auto-generated code in HiDP by a large margin. HiDP does not intend to compete with well-refined numerical libraries. But what HiDP shows is that different code transformation strategies are suited for different data inputs, even on the same machine. It is necessary to emit a complete selection of alternatives for auto-tuning or for users to choose from.

6.4.2 3D Stencil Computation

```
function himeno
input float a0[I][J][K], a1[I][J][K], a3[I][J][K], b0[I][J][K], b1[I][J][K], b2[I][J][K], c0[I][J][K], c1[I][J][K], c2[I][J][K], p[I
    ][J][K], wrk1[I][J][K], bnd[I][J][K], _omega;
output float _gosa, wrk2[I][J][K];
{
  map{
    i:=[1:I−2]; j:=[1:J−2]; k:=[1:K−2];
    _s0 = _a0∗ p[i+1][j][k] + _a1∗ p[i][j+1][k] + _a2∗ p[i][j][k+1] + _b0∗(p[i+1][j+1][k] − p[i+1][j−1][k] − p[i−1][j+1][k]
        + p[i−1][j−1][k]) + _b1∗(p[i][j+1][k+1] − p[i][j+1][k−1] − p[i][j−1][k+1] + p[i][j−1][k−1]) + _b2∗(p[i+1][j][k
        +1] − p[i+1][j][k−1] − p[i−1][j][k+1] + p[i−1][j][k−1]) + _c0∗ p[i−1][j][k] + _c1∗ p[i][j−1][k] + _c2∗ p[i][j][k
        −1] + _wrk1;
    _ss = (_s0 ∗ _a3 − _p) ∗ _bnd;
    _wrk2 = p + _omega ∗ _ss;
    _ss2 = _ss ∗ _ss;
  }reduce('+', _gosa, _ss2, i:=[∗], j := [∗], k:=[∗]);
}
```

Figure 6.7: Himeno Benchmark in HiDP

An interesting group of computations that are well-suited for GPUs are Jacobi stencil computations [43]. In stencil computation, new values of elements are updated based on old values of the local element and their neighbors. There are different neighbor access patterns for different types of stencil computation. HiDP detects such patterns and optimizes them using on-chip Shared Memory to save off-chip memory bandwidth.

We select two stencil computations (7-point and the Himeno benchmark) utilizing double-precision floating-point and compare the performance of HiDP generated code with another adaptive framework

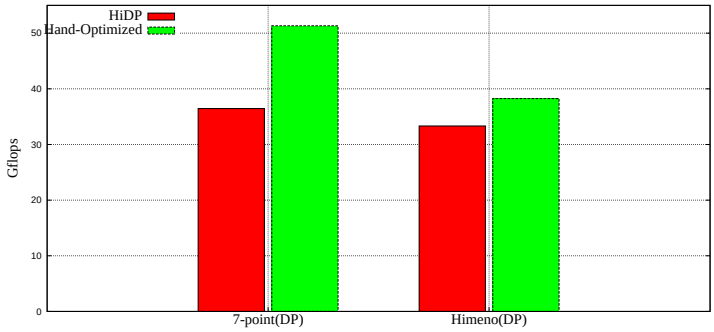


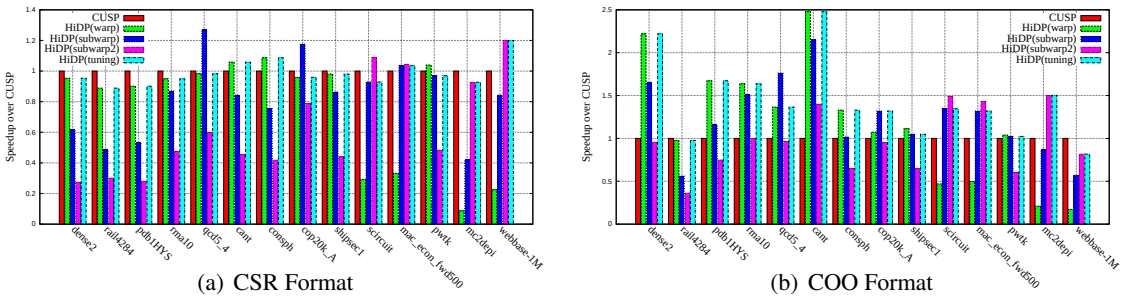
Figure 6.8: Comparing HiDP with Auto-Tuned Code in Stencil Computations

for stencil computations [125]. The details of the Himeno computation can be found in [85, 96]. Figure 6.7 shows how Himeno is expressed in HiDP. The main part is a map clause with a reduction suffix. The map clause gives the user customized control of map iterators, which exclude boundary indices in all three axes here. Array accesses are assumed to be in the order of the declarations of map iterators. Brackets can be omitted if the access is independent per thread. For example, $\_a0$ inside the map clause means $a0[i][j][k]$. We can see that HiDP is almost as concise as the domain specific language in [125]. By adding a reduction using the map suffix, HiDP can even generate reduction code inside the same kernel as the stencil computation, a feature that is not available in other frameworks [125].

Because there is only a single-level map in HiDP and the reduction is applied to all map iterators (a global reduction), the compiler selects a kernel-level execution mode where the entire map clause becomes a CUDA kernel. The reduction at this kernel-level mode is a two-phase process involving both the GPU and CPU: each block performs a block-level reduction and then the CPU reduces all local reduction values into a single one.

The GFlops difference is shown in Figure 6.8. HiDP generates a block size of 16×16 by default. In contrast, [125] uses off-line tuning to search for the best parameters for a stencil. This difference contributes to the performance difference between them. However, HiDP still manages to reach at least 70% of the GFlops performance.

6.4.3 Sparse Matrix Vector Multiplication



(a) CSR Format (b) COO Format

Figure 6.9: Sparse Matrix Vector Multiply

A typical sparse matrix vector multiplication routine has a two-level map block where the outer level iterates over each row of the sparse matrix and the inner level iterates over each element in the same row and then performs a reduction on this row. The shape analysis generates a two-level nested shape $\{[0 : row), *\}$ for the second-level map clause. The $*$ indicates that the parallelism degree is data dependent. This uncertainty, if not further constrained by the user, leads the HiDP compiler to

try several options to determine which execution mode to choose at the second level. The number of rows is used by HiDP as the parallelism degree, *i.e.*, it determines which mapping has a better chance to utilize all GPU computing resources. In the following, we show results obtained by three decisions where the inner map is executed (1) by an entire warp, (2) by a subwarp of 8 threads (subwarp) and (3) by a subwarp of 4 threads (subwarp2). As the number of rows for the sparse matrix increases, HiDP tends to use less threads per inner map.

Figure 6.9 depicts the speedups achieved for each choice using a hand-written sparse matrix vector library (CUSP) as the baseline. On the X axis, matrices are ordered by increasing number of rows from left to right. We see a clear performance benefit of using fewer threads per row as the number of rows increases, with only a few exceptions near the switching point. (In COO format, HiDP still chooses subwarp as the parallelization alternative for the scircuit matrix — even though subwarp2 is slightly faster. This is due to a significant performance loss of subwarp2 for the pwtk matrix.) With our tuning capability, HiDP delivers very close or even better performance than hand-written CUDA code.

Another observation is that HiDP performs better in COO format than CSR format in general. This is due to implementation differences between our HiDP code and CUSP for the COO format: HiDP uses an auxiliary array to convert the COO format into the CSR format and reuses the CSR kernels. In contrast, CUSP performs segmented reduction for the COO format, which turns out to be slower.

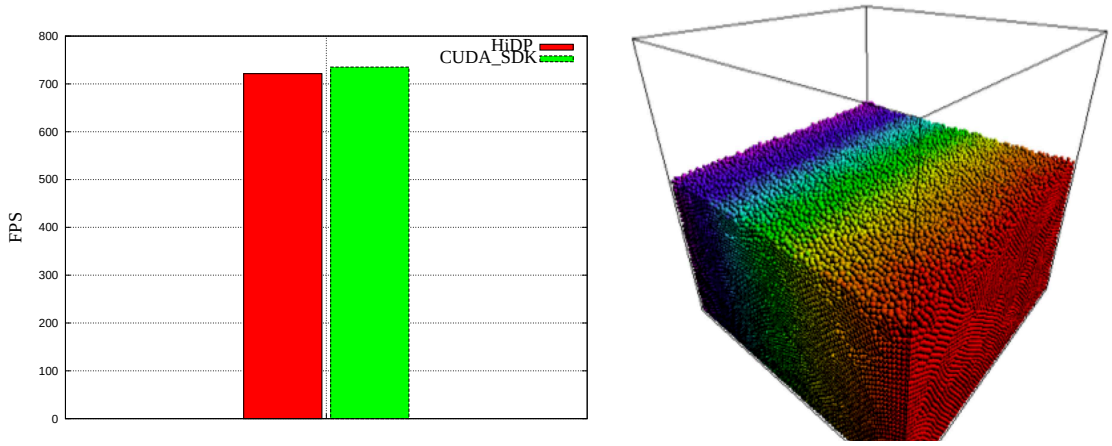
Just using the number of rows to determine the switching point is by no means optimal. The distribution characteristics (min, max and average etc.) of the number of non-zero entries in each row of the sparse matrix should affect the decision, too. The HiDP compiler, at this point, does not consider these aspects for more advanced decisions. If equipped with prior knowledge, the user has to manually choose the appropriate implementation.

6.4.4 Particle Simulation

As a demonstration of a pipeline of kernels, we choose the particle simulation example of the CUDA SDK. We simulate collisions of 128K particles in a cube (Figure 6.10(b)). The core of the simulation consists of a sequence of steps: an update of particle velocities and positions, hashing, sorting and collision detection. After rewriting the algorithm into a much more concise HiDP code, the compiler emits several kernels similar to the hand-written code of the CUDA SDK. As a result, the FPS (frame per second) metric shows little difference (Figure 6.10(a)).

6.4.5 Quicksort

It is very easy to express quicksort in HiDP because HiDP supports segmented *partition* and *sort* as built-in parallel primitives (see Table 6.2). In HiDP, quicksort performs a few iterations of partitioning with carefully chosen pivots. (We use the average of the min and max.) In the beginning, there is only one segment. The number of segments doubles after each iteration. After the number of segments is



(a) FPS Comparison (b) Particle Simulation

Figure 6.10: Particle Simulation

large enough (64 or more), we finish with segmented sort in a separate map clause, which internally uses a merge sort implementation.

We compare HiDP's code with GPU-Quicksort, a hand-written CUDA sorting library using quicksort [27]. GPU-Quicksort also starts with partitioning an array into smaller segments but then switches to bitonic sort. To the best of our knowledge, it is the fastest open-source GPU implementation utilizing quicksort.

Figure 6.11 depicts the execution time of each implementation for 4 to 32 million unsigned integers. The input distribution is uniform. (We observed similar patterns for other input distributions.) HiDP is able to keep up with GPU-Quicksort in terms of performance. But in contrast to GPU-Quicksort, HiDP shines in coding productivity: the total number of source code lines for GPU-Quicksort, including both

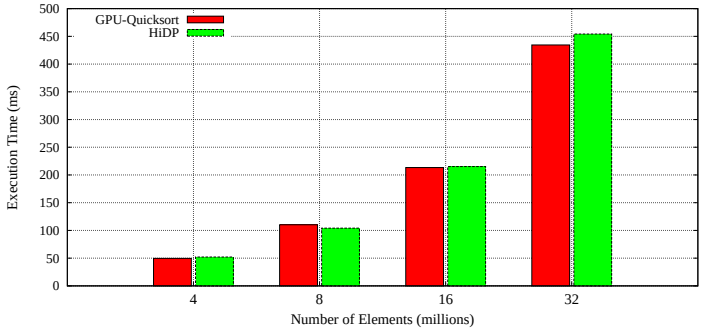


Figure 6.11: Quicksort

host-side C++ and CUDA, adds up to about 900 lines. The equivalent HiDP code is just short of 50 lines, more than an order of magnitude less.

6.4.6 Bitonic Sort

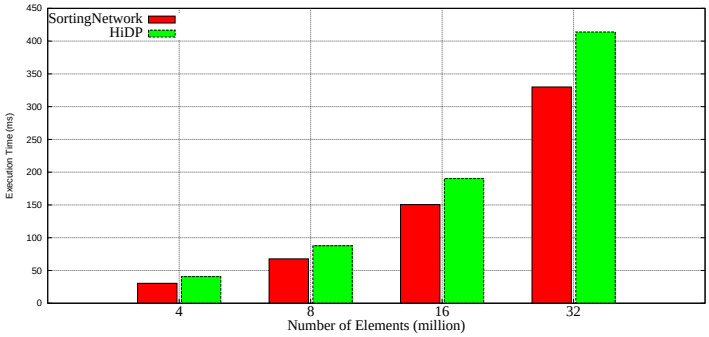


Figure 6.12: Bitonic Sort

Bitonic sort is a good show case for HiDP's support of a regular interface for parallel primitives because it always works on segments of the same size and the total size has to be a power of two. Here, we compare the performance with that of the same algorithm released by Nvidia in the CUDA SDK. In this example, auto-generated HiDP code achieves up to 80% of the performance of hand-written code (Figure 6.12). Similar to quicksort, bitonic sort in HiDP requires only $\approx$50 lines of code. In contrast, the hand-written CUDA SDK requires more than 250 lines of code.

6.5 Related Work

There are numerous propositions to extend existing languages with directional annotations. Lee *et al.* were the first to support CUDA code generation with OpenMP annotations ([79, 78]). The StarSs programming model represents a group of variants (OmpSs [45], GpuSs [16]) under a common theme: exploiting task-level parallelism via compiler pragmas on task arguments. It relies on the read/write properties of task arguments to build a task dependency graph and creates necessary memory copies. StarPU [14] offers a unified task abstraction. Tasks can be implemented by "codelet", which targets different architectures. Both StarSs and StarPU focus on run-time scheduling of tasks and do not alleviate users from writing low-level kernels. HMPP [99] and OpenACC [74] are recent approaches to utilize OpenMP-like pragmas on parallelizable code sections, which are often do/for loops. Their optimization scope is limited to block level and they both lack auto-tuning capabilities.

On the language side, Sequoia [50] adds memory hierarchy as a first class feature in its language with task parallelism. It captures the importance of utilizing the memory hierarchy of modern architectures, which is also part of HiDP's optimization strategies. Chapel [29] and UPC [46] are parallel languages for a Single Program Multiple Data (SPMD) model of parallelism. They provide high degree of programmability with a global address space. Chapel also supports nested parallelism with mixed task and data parallelism. C++ AMP [40] extends C++ to support data-parallel accelerators. Nested parallel for loops are absent from C++ AMP, for it treats the underling accelerator as a flat architecture — unless the user uses language extensions to write kernels in a similar manner to CUDA or OpenCL. None of the above languages exploit the hierarchical execution model of GPUs to the extend that HiDP does.

The Petabricks compiler [10] provides an encapsulation of a function body that is similar to HiDP's approach. This modular design is convenient for compilers with auto-tuning capabilities. In contrast, Petabricks's tuning is for algorithmic choices. Users need to provide native code for each algorithm.

An active research topic is source-to-source compiler framework that translates well-established high-level languages (data-parallel Haskel, Python) to CUDA. Garg and Amaral [53] propose compiling techniques to convert Python loop structures and array operations to CUDA code. But to stay efficient, they require the programmer to conform to the style of the targeted language (similar to C++ AMP). Copperhead [26] conforms to Python's syntax as much as possible without introducing codelets. It advocates the mapping of nested parallel structures into a hierarchical execution model. Though this mapping can be directed by the end-user, it is static and lacks the tuning capabilities that are essential for performance, as shown in our work. Even though HiDP provides similar functionality to CuNesl (Chapter 5 and [126]), HiDP tends to have better performance because the standard flattening method to convert nested parallelism into the segmented counterpart (what the CuNesl does) is too general an approach that fails to efficiently utilize the hierarchical resources of GPUs.

Overall, HiDP provides a low-level, hierarchical STL-like interface with data-parallel language features. The user can concentrate on algorithmic design while benefiting from the hand-crafted common primitives developed by architecture experts. The single-entry function design helps to integrate HiDP with an existing mixed language code base.

6.6 Future Work

HiDP is under active development and is subject to many improvements in the future. Some of the ideas are:

- More aggressive and efficient usage of Shared Memory is desirable considering Shared Memory is a scarce resource. This includes better detection of shared patterns between neighboring threads and time-sharing Shared Memory storage between different data-parallel primitives within the

same kernel.

- Currently, HiDP's auto-tuning focuses on kernel selection. Configurations for each kernel (block sizes *etc.*) are heuristically chosen and kept constant. However, data types and algorithms affect the usage of registers and Shared Memory, which in turn affect the utilization ratio. Smarter choices for block sizes or being able to tune them will be important to shorten the performance gap between auto-generated code and handwritten one.

- HiDP does not support new features of forthcoming GPU generations. An interesting feature is the support of dynamic parallelism [94] in Nvidia's Kepler architecture (the forthcoming K20 as K10 is still lacking this feature). It has yet to be see be seen how the language needs to adjust to support that.

- Adding more backend support is always desirable for increasing the impact of HiDP. A more portable OpenCL is a compelling target.

6.7 Conclusion

Inserting directional annotations to legacy code may be a desirable method to take advantage of new compiling and architecture features, yet such annotations limit the optimization space, and their applicability is often restricted to only selected algorithms. In practice, data structures and algorithms tend to require changes to better utilize computational resources of modern parallel architectures. High-level languages that embrace performance efficiency and coding productivity seem to be a more promising solution to improve performance.

This chapter presents HiDP, a hierarchical data-parallel language designed for efficient execution on today's SIMT architectures. HiDP allows users to express algorithms as a mixture of both task-level and data-level parallelism. HiDP's compiler performs kernel fusion based on symbolic shape analysis and integrates with common handwritten data-parallel primitives. HiDP explores various execution mappings according to the structures of the application and searches for appropriate dynamic switching points via auto-tuning.

The motivation of using HiDP reaches beyond coding productivity. Our experimental results show that HiDP is capable of achieving good performance for many types of applications compared to their hand-written counterparts. HiDP is an active project with an forthcoming open-source release.

Chapter 7

Conclusion

In this chapter, we summarize previous chapters and present the conclusion of this dissertation.

We start from hand coding a large-scale application, namely document clustering using flocking based simulation, on current GPU clusters. By carefully redesigning algorithms and data structures, we can take advantage of the massive parallel computing resources inside GPUs and obtain impressive speedups over CPUs of the same size. However, this is achieved with substantial hand-tuning efforts.

We then move to designing programming model and run-time systems to reduce the programming effort on GPU clusters. We begin with domain specific areas such as streaming applications. We propose GStream, a scalable data streaming framework. It contains concise language abstractions to help users express data operations and data movement. Another domain specific approach is a language front-end and compiling framework for Jacobi-style stencil computations. It outputs highly-efficient CUDA/MPI code that can be tuned to achieve optimal performance over the designed tuning space.

As the first solution for handling more general-purpose applications, we design a compiler to convert NESL, an existing nested data parallel languages, into CUDA code. We develop techniques to convert recursive calls into while loops with the help of segmented arrays.

However, implementing nested parallelism with segmented structures is not ideal for today's hierarchical architectures. To shorten the performance gap between generated code and handwritten code, we propose HiDP, a hierarchical data-parallel language with nested parallel-for structures. It also supports seamless integration with efficient data-parallel primitives. Experimental results show that the proposed framework not only improves coding productivity substantially but also proves to be competitive compared to handwritten code.

All of the approaches above demonstrate that programming with data parallelism is mandatory today to fully utilize the increasing number of computing cores in state-of-the-art microprocessors. Our work touches various areas of the programming tool chain to better exploit data parallelism. This, in turn, provides improved programmability and performance for microprocessor architectures, thus validating the hypothesis of this dissertation.

REFERENCES

[1] CUDA CUBLAS library.

[2] CUDA CUFFT library.

[3] http://en.wikipedia.org/wiki/stop_words.

[4] http://loki-lib.sourceforge.net/.

[5] http://math.nist.gov/tnt/.

[6] http://www.nvidia.com/object/cuda_home.html.

[7] NVIDIA Cooperation, CUDA Programming Guide.

[8] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[9] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[10] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[11] Clint Whaley Antoine, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27:2001, 2000.

[12] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. Technical Report 2004-20, Stanford InfoLab, 2004.

[13] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB*, pages 480–491. Morgan Kaufmann, 2004.

[14] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. NAS Parallel Benchmark Results. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[18] Systems Michael Beynon, Michael Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman, Joel Saltz, and Johns Hopkins Medical. DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage. In *In IEEE Symposium on Mass Storage Systems*, pages 119–133. IEEE Computer Society Press, 2000.

[19] Guy Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 21:102–111, 1994.

[20] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[21] Guy E Blelloch and Siddhartha Chatterjee. VCODE: A Data-Parallel Intermediate Language. In *Proceedings Frontiers of Massively Parallel Computation*, pages 471–480, 1990.

[22] Guy E Blelloch and Parallel Ram Model. NESL: A Nested Data-Parallel Language. Technical report, 1993.

[23] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM TRANSACTIONS ON GRAPHICS*, 23:777–786, 2004.

[24] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a Declarative Language for Real-Time Programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[25] Bryan Catanzaro, Armando Fox, Kurt Keutzer, David Patterson, Bor-Yiing Su, Marc Snir, Kunle Olukotun, Pat Hanrahan, and Hassan Chafi. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. *IEEE Micro*, 30(2):41–55, March 2010.

[26] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[27] Daniel Cederman and Philippas Tsigas. A Practical Quicksort Algorithm for Graphics Processors. In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 246–258, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a Status Report. In *Proceedings of the 2007 workshop on*

Declarative aspects of multicore programming, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM.

[29] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[30] Roger D. Chamberlain, Mark A. Franklin, Eric J. Tyson, James H. Buckley, Jeremy Buhler, Greg Galloway, Saurabh Gayen, Michael Hall, E.F. Berkley Shands, and Naveen Singla. Auto-pipe: Streaming applications on architecturally diverse systems. *Computer*, 43:42–49, 2010.

[31] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[32] Jesse St. Charles, Thomas E. Potok, Robert M. Patton, and Xiaohui Cui. Flocking-based Document Clustering on the Graphics Processing Unit. *NICSO*, pages 27–37, 2007.

[33] Siddhartha Chatterjee. Compiling Nested Data-Parallel Programs for Shared-Memory Multiprocessors. *ACM Trans. Program. Lang. Syst.*, 15:400–462, July 1993.

[34] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[35] Gang Chen, Guobo Li, Songwen Pei, and Baifeng Wu. GPGPU Supported Cooperative Acceleration in Molecular Dynamics. *International Conference on Computer Supported Cooperative Work in Design*, 0:113–118, 2009.

[36] Tong Chen and Zehra Sura. Optimizing the use of static buffers for dma on a cell chip. In *In The 19th International Workshop on Languages and Compilers for Parallel Computing*, 2006.

[37] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uur etintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003.

[38] Francisco Chinchilla, Todd Gamblin, Morten Sommervoll, and Jan F Prins. Parallel N-Body Simulation Using GPUs. Technical report, University of North Carolina at Chapel Hill, 2004.

[39] Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures. *In IEEE Intl Parallel and Distributed Processing Symposium (IPDPS)*, May 2011.

[40] Microsoft Cooperation. C++ AMP : Language and Programming Model. 2012.

[41] Xiaohui Cui, Jinzhu Gao, and Thomas E. Potok. A Flocking Based Algorithm for Document Clustering Analysis. *J. Syst. Archit.*, 52(8):505–515, 2006.

[42] Matthew Curry, Lee Ward, Tony Skjellum, and Ron Brightwell. Accelerating Reed-Solomon Coding in RAID Systems with GPUs. In *IPDPS*, apr 2008.

[43] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Autotuning on State-of-the-art Multicore Architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[44] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-Based PDE Solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[45] Alejandro Duran, Eduard Ayguad, Rosa M. Badia, Jess Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, pages 173–193, 2011.

[46] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[47] Ugo Erra, Rosario De Chiara, Vittorio Scarano, and Maurizio Tatafiore. Massive Simulation Using GPU of a Distributed Behavioral Model of a Flock with Obstacle Avoidance. In *Proceedings of Vision, Modeling and Visualization 2004 (VMV)*, November 2004.

[48] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[49] Wenbin Fang, Ka K. Lau, Mian Lu, Xiangye Xiao, Chi K. Lam, Philip Y. Yang, Bingsheng He, Qiong Luo, Pedro V. Sander, and Ke Yang. Parallel Data Mining on Graphics Processors. Technical report, The Hong Kong University of Science and Technology, October 2008.

[50] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[51] Massimiliano Fatica and Won-Ki Jeong. Accelerating MATLAB with CUDA. In *HPEC*, September 2007.

[52] Matteo Frigo. A Fast Fourier Transform Compiler. *SIGPLAN Not.*, 39:642–655, April 2004.

[53] Rahul Garg and José Nelson Amaral. Compiling Python to a Hybrid Execution Environment. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 19–30, New York, NY, USA, 2010. ACM.

[54] Thierry Gautier, Paul Le Guernic, and Löic Besnard. SIGNAL: A Declarative Language for Synchronous Programming of Real-Time Systems. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 257–277, London, UK, 1987. Springer-Verlag.

[55] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[56] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast Computation of Database Operations Using Graphics Processors. In *Proc. of ACM SIGMOD*, pages 215–226. ACM Press, 2004.

[57] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast Computation of Database Operations Using Graphics Processors. In *SIGMOD '04*, pages 215–226, New York, NY, USA, 2004. ACM.

[58] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors. In *SIGMOD '05*, pages 611–622, New York, NY, USA, 2005. ACM.

[59] Portland Group. PGI CUDA Fortran Compiler.

[60] Ping Guo and Liqiang Wang. Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs. *Computational and Information Sciences, International Conference on*, 0:1154–1157, 2010.

[61] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31:532–533, May 1988.

[62] Tianyi David Han and Tarek S. Abdelrahman. hicuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM.

[63] Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *HiPC*, pages 197–208, 2007.

[64] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[65] Jared Hoberock and Nathan Bell. Thrust: A Parallel Template Library, 2010. Version 1.3.0.

[66] Kenneth E. Hoff, III, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *Proceedings of the sixteenth annual symposium on Computational geometry*, SCG '00, pages 375–376, New York, NY, USA, 2000. ACM.

[67] http://code.google.com/p/cudpp/. CUDPP.

[68] http://www.accelereyes.com. Jacket.

[69] http://www.khronos.org/opencl. OpenCL.

[70] Wen-mei Hwu, Shane Ryoo, Sain-zee Ueng, John H. Kelm, Isaac Gelado, Sam S Stone, Robert E. Kidd, Sara S Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly Parallel Programming Models for Thousand-core Microprocessors. In *Proceedings of the 44th annual Design Automation Conference*, pages 754–759, New York, NY, USA, 2007. ACM.

[71] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 431–442, New York, NY, USA, 2006. ACM.

[72] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *In IEEE Intl Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[73] Mark Kantrowitz, Behrang Mohit, and Vibhu Mittal. Stemming and its effects on tfidf ranking (poster session). In *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 357–359, New York, NY, USA, 2000. ACM.

[74] Axel Koehler. GPU Programming with CUDA and OpenACC. 2012.

[75] Jens Krüger and Rüdiger Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 908–916, New York, NY, USA, 2003. ACM.

[76] Orion S. Lawlor. Message Passing for GPGPU Clusters: CudaMPI. In *CLUSTER*, pages 1–8. IEEE, 2009.

[77] Sean Lee, Vinod Grover, Manuel M. T. Chakravarty, and Gabriele Keller. Gpu kernels as data-parallel array computations in haskell, 2009.

[78] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysi s*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[79] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44:101–110, February 2009.

[80] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. In *In Proc. SIGGRAPH*, pages 327–335, 1990.

[81] Yinan Li, Jack Dongarra, and Stanimire Tomov. A Note on Auto-tuning GEMM for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[82] Zhiyuan Li and Yonghong Song. Automatic Tiling of Iterative Stencil Loops. *ACM Trans. Program. Lang. Syst.*, 26:975–1028, November 2004.

[83] William R. Mark, R. Steven, Glanville Kurt, Akeley Mark, and J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics*, 22:896–907, 2003.

[84] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. 2011.

[85] Satoshi Matsuoka, Takayuki Aoki, Toshio Endo, Akira Nukada, Toshihiro Kato, and Atushi Hasegawa. GPU Accelerated Computing from Hype to Mainstream, the Rebirth of Vector Computing. In *Journal of Physics: Conference Series 180*, 2009.

[86] Jiayuan Meng and Kevin Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[87] Paulius Micikevicius. 3D Finite Difference Computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[88] S. Momen, B.P. Amavasai, and N.H. Siddique. Mixed Species Flocking for Heterogeneous Robotic Swarms. In *EUROCON, 2007. The International Conference on "Computer as a Tool"*, pages 2329–2336, Sept. 2007.

[89] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Resource Management, and Approximation ina Data Stream Management System. Technical Report 2002-41, Stanford InfoLab, 2002.

[90] Anurag Acharya Mustafa, Mustafa Uysal, and Joel Saltz. Active Disks: Programming Model, Algorithms and Evaluation, 1998.

[91] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[92] Hubert Nguyen(ed). *GPU Gems 3*. Addison-Wesley Professional, 2007.

[93] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[94] Nvidia. Kepler GK110 Whitepaper. 2012.

[95] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive Multi-pass Programmable Shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 425–432, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[96] E.H. Phillips and M. Fatica. Implementing the Himeno benchmark with CUDA on GPU clusters. In *International Parallel and Distributed Processing Symposium(IPDPS)*, Apr 2010.

[97] Steve Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117:1–19, 1995.

[98] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. pages 703–712, 2002.

[99] S. Bihan R. Dolbeau and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, Oct 2007.

[100] Joel W. Reed, Yu Jiao, Thomas E. Potok, Brian A. Klump, Mark T. Elmore, and Ali R. Hurson. TF-ICF: A New Term Weighting Scheme for Clustering Dynamic Data Streams. In *ICMLA '06: Proceedings of the 5th International Conference on Machine Learning and Applications*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[101] Craig Reynolds. Steering Behaviors for Autonomous Characters. In *Game Developers Conference*, 1999.

[102] Craig W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics*, 21(4):25–34, 1987.

[103] Konrad Rieck and Pavel Laskov. Linear-Time Computation of Similarity Measures for Sequential Data. *J. Mach. Learn. Res.*, 9:23–48, 2008.

[104] Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. Cactus Application: Performance Predictions in Grid Environments. In *In proceedings of European Conference on Parallel Computing (EuroPar) 2001*, 2001.

[105] Antonio J. Rueda Ruiz and Lidia M. Ortega. Geometric Algorithms on CUDA. In *GRAPP*, pages 107–112, 2008.

[106] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[107] Jay Sipelstein and Guy E Blelloch. Collection-Oriented Languages. *Proceedings of the IEEE*, 79(4):504–523, 1991.

[108] Michael Steinbach, George Karypis, and Vipin Kumar. A Comparison of Document Clustering Techniques, 2000.

[109] R. Stephens. A survey of stream processing, 1995.

[110] Jeff A. Stuart and John D. Owens. Message Passing on Data-parallel Architectures. In *IPDPS*, pages 1–12, 2009.

[111] Michael Beynon Tahsin, Michael D. Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Design of a Framework for Data-Intensive Wide-Area Applications. In *In Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000*, pages 116–130. IEEE Computer Society Press, 2000.

[112] G. Teodoro, R. Sachetto, O. Sertel, M.N. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira. Coordinating the Use of GPU and CPU for Improving Performance of Compute Intensive Applications. *IEEE International Conference on Cluster Computing and Workshops*, pages 0–10, 2009.

[113] Bill Thies, Michal Karczmarek, and Saman Amarasinghe. StreaMIT: A Language for Streaming Applications. In *In International Conference on Compiler Construction*, pages 179–196, 2001.

[114] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software Pipelined Execution of Stream Programs on GPUs. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 200–209, Washington, DC, USA, 2009. IEEE Computer Society.

[115] Didem Unat, Xing Cai, and Scott Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *Proceedings of the 25th International Conference on Supercomputing (ICS'11)*, 2011.

[116] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33:103–111, August 1990.

[117] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A library of Automatically Tuned Sparse Matrix Kernels. In *Institute of Physics Publishing*, 2005.

[118] William W. Wadge and Edward A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[119] Ren Wu, Bin Zhang, and Meichun Hsu. Clustering Billions of Data Points Using GPUs. In *UCHPC-MAW '09: Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 1–6, New York, NY, USA, 2009. ACM.

[120] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[121] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.

[122] Sain zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen mei W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In *LCPC08*, 2008.

[123] Erik Zeitler and Tore Risch. Scalable splitting of massive data streams. In *DASFAA, Proc. 15th Conf. on Database Systems for Advanced Application*, 2009.

[124] Y. Zhang, F. Mueller, Xiaohui Cui, and Thomas Potok. Gpu-accelerated text minining. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, March 2009.

[125] Yongpeng Zhang and Frank Mueller. Auto-Generation and Auto-Tuning of 3D Stencil Codes on GPU Clusters. *In International Symposium on Code Generation and Optimization (CGO)*, April 2012.

[126] Yongpeng Zhang and Frank Mueller. CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures. *In International Conference on Parallel Processing (ICPP)*, Sep 2012.

[127] Bo Zhou and Suiping Zhou. Parallel Simulation of Group Behaviors. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 364–370. Winter Simulation Conference, 2004.

[128] Jin Zhou and Brian Demsky. Bamboo: a Data-Centric, Object-Oriented Approach to Many-core Software. *SIGPLAN Not.*, 45:388–399, June 2010.