Understanding the Tradeoffs between Software-Managed vs. Hardware-Managed Caches in GPUs

Chao Li
Department of Electrical and Computer Engineering
North Carolina State University
cli17@ncsu.edu

Yi Yang
Department of Computing Systems Architecture
NEC Laboratories America
yyang@nec-labs.com

Hongwen Dai
Department of Electrical and Computer Engineering
North Carolina State University
hdai3@ncsu.edu

Shengen Yan
Institute of Software, Chinese Academy of Sciences,
North Carolina State University
yanshengen@gmail.edu

Frank Mueller
Department of Computer Science
North Carolina State University
fmuelle@ncsu.edu

Huiyang Zhou
Department of Electrical and Computer Engineering
North Carolina State University
hzhou@ncsu.edu

***Abstract*—On-chip caches are commonly used in computer systems to hide long off-chip memory access latencies. To manage on-chip caches, either software-managed or hardware-managed schemes can be employed. State-of-art accelerators, such as the NVIDIA Fermi or Kepler GPUs and Intel's forthcoming MIC "Knights Landing" (KNL), support both software-managed caches, aka. shared memory (GPUs) or near memory (KNL), and hardware-managed L1 data caches (D-caches). Furthermore, shared memory and the L1 D-cache on a GPU utilize the same physical storage and their capacity can be configured at runtime (same for KNL). In this paper, we present an in-depth study to reveal interesting and sometimes unexpected tradeoffs between shared memory and the hardware-managed L1 D- caches in GPU architecture. In our study, the kernels utilizing the L1 D-caches are generated from those leveraging shared memory to ensure that the same optimizations such as tiling are applied equally in both versions. Our detailed analyses reveal that rather than cache hit rates, the following tradeoffs often have more profound performance impacts. On one hand, the kernels utilizing the L1 caches may support higher degrees of thread-level parallelism, offer more opportunities for data to be allocated in registers, and sometimes result in lower dynamic instruction counts. On the other hand, the applications utilizing shared memory enable more coalesced accesses and tend to achieve higher degrees of memory-level parallelism. Overall, our results show that most benchmarks perform significantly better with shared memory than the L1 D-caches due to the high impact of memory-level parallelism and memory coalescing.**

I. INTRODUCTION

To manage on-chip caches effectively, either explicit software management or implicit hardware management schemes have been widely used in computer systems. While hardware-managed caches relieve the application developers of explicit data management, it is expected that software approaches may offer higher cache performance (i.e., hit rates) with the knowledge of data reuse patterns. State-of-art graphics processing units (GPUs), such as the NVIDIA GTX480 and GTX680 GPUs, include both software managed caches, aka. shared memory, and hardware managed L1 data caches (D-caches). An outstanding feature of these GPUs is that shared memory and L1 D-caches utilize the same physical resource and their capacities can be configured through APIs. As a result, GPUs provide an ideal platform to study the intriguing tradeoffs between hardware-managed caches and software-managed caches.

In this paper, we aim to provide insights to the following questions: (a) is it worthwhile for application developers to explicitly manage shared memory with the existence of the hardware managed L1 D-caches in GPUs? And (b) what are the main reasons for code utilizing shared memory to outperform code leveraging L1 D-caches (and vice versa)?

We start our journey with the well-known matrix multiplication algorithm. From an optimized kernel using shared memory, we remove shared memory arrays while ensuring that the same tiling optimization has been applied. As the tiles fit in the D-cache/shared memory capacity, both versions enjoy very high hit rates after the tiles are initially loaded into shared memory/the L1 D-cache. The measured execution times on GTX480 GPUs (GTX680 GPUs do not cache global memory data in L1 D-caches), however, show that the L1 D-cache version is surprisingly much slower (43.8%) than the shared-memory version. Puzzled with these results, we resort to a GPU architectural timing simulator, which reports a similar performance trend, to retrieve detailed execution statistics. Through micro-kernels, assembly-level code analysis, and cycle-by-cycle instruction execution information, we pinpoint the unexpected reason: the shared-memory version achieves much higher memory-level parallelism (MLP) than the L1 D-cache version. Other factors including hit rates, memory coalescing, and dynamic instruction counts, have relatively little impact in comparison.

Besides matrix multiplication, we also perform detailed case studies on FFT, MC, and PF due to their distinctive data access patterns. Then, we categorize a set of 16 GPGPU workloads based on whether or not there is data sharing among threads. Our results show that for most applications, the GPU kernels utilizing shared memory deliver significantly higher

performance than those leveraging L1 D-caches. The fundamental reasons are MLP and coalescing. For a few benchmarks for which the L1 D-cache versions have higher performance, the performance impact is mainly due to improved thread-level parallelism (TLP) and allocating more data to registers. Overall, rather than cache hit rates, the subtle factors including MLP, coalescing, and TLP often have more profound performance impacts.

The remainder of the paper is organized as follows. Section II presents a brief background on GPU computing with an emphasis on GPU memory hierarchy. Section III contains four detailed case studies to reveal the interesting tradeoffs between shared memory and L1 D-caches. Section IV categorizes the benchmarks based on their data access patterns and discusses their tradeoffs in utilizing shared memory or L1 D-caches. Section V addresses the related work. Section VI concludes.

II. BACKGROUND

State-of-art GPUs leverage high degrees of TLP to achieve high computational throughput. In GPU programming models such as NVIDIA CUDA [5], a GPU kernel is launched as a grid of thread blocks (TBs) and each TB in turn contains many threads. During execution, the threads in a TB are grouped into multiple warps based on their thread identifiers (ids) and threads in a warp execute in a lockstep manner, meaning that there is one program counter (pc) for all the threads in a warp.

Although TLP is key to hide long memory-access latencies, the GPU memory hierarchy remains critical to many GPU applications. The GPU memory space consists of texture memory, constant memory, local memory, shared memory, and global memory. The texture and constant memory are for read-only data and are accessible to all the threads in different TBs. Global memory can be read and written to by all the threads. In contrast, local memory is private for each thread while shared memory contains data that are local to each thread block. Shared memory is physically located on-chip, providing low access latencies and high bandwidths. As shared-memory variables are explicitly defined and used in GPU programs, it is a software-managed cache, similar to the local store in the CellBE architecture and scratchpad memory used in embedded systems. In comparison, hardware-managed caches are used for different types of GPU memory. Besides dedicated read-only caches for texture and constant memory, recent GPUs, including the NVIDIA Fermi [10] and Kepler [11] architectures, provide a level-one (L1) D-cache in each streaming multiprocessor (SM). In the Fermi architecture, the L1 data cache is used for both local and global memory data. In Kepler architecture, the L1 data cache is used only for local memory data. To simplify the cache-coherence management, the L1 caches employ a write-evict policy for global memory data in the Fermi architecture. Besides the L1 D-caches, there is a unified L2 cache shared among multiple SMs. Due to the limited capacity of shared memory in each SM, one side effect of shared-memory usage is that it may limit the number of TBs that can run concurrently in an SM.

III. TRADEOFFS BETWEEN SHARED MEMORY AND L1 D-CACHES

A. Case Study I: Matrix Multiplication

For matrix multiplication with large matrices, the key optimization is loop tiling/blocking to reduce the working set size and thereby reduce reuse distances. The matrix multiplication kernel from CUDA SDK [12] uses shared memory to store the tiles and the code is shown in Figure 1a.

The code in Figure 1a computes a tile of elements in the product matrix. It reads a tile from either of the two input matrices and stores the two tiles into shared memory. Then, it performs multiplication between the two tiles before moving on to the next set of tiles. Since the tiles already reside in shared memory when multiplication is performed, the accesses to the tiles have low latency, thereby achieving high performance. This code illustrates the explicit way of managing data, which dictates where the data are stored and where to access them. It also showcases an overhead of shared-memory usage. The data are loaded to registers first and stored to shared memory. The same data will then be loaded again from shared memory into registers to perform computation. Such redundant data accesses result in increased dynamic instruction counts.

When an L1 D-cache is used, the tiles are stored in cache implicitly. When one element is loaded for computation, its neighbors in the same cache block/cache line will be loaded into cache. It does not need additional instructions to do so. Subsequent accesses, when hitting in cache, will load data from the cache directly. The code for the matrix multiplication kernel using the L1-D-cache is shown in Figure 1b. From the Figure 1a and Figure 1b, we can see that both versions of code use exactly the same tiling technique. The difference between the two versions is that in the inner loop, the shared-memory version accesses the data from shared memory explicitly while the D-cache version accesses the data from global memory, therefore the cache implicitly. For the shared-memory version, we set the shared memory capacity as 48kB on a GTX480 GPU. For the D-cache version, we configure the L1 D-cache capacity as 48kB on the same GPU. The input matrices have a size of 256x256 and the tile size is 16x16. Therefore, each tile has a size of 1kB (=16x16x4B). With either configuration, the register usage is the limiting factor on how many TBs can run concurrently on an SM. For both versions, 5 TBs can run concurrently on a TB. Since the L1 D-cache and shared memory actually use the same physical on-chip storage and the capacity is large enough for the working sets (10kB for 5TBs), our initial expectation is the two versions should have similar performance. If we consider the instruction overhead for explicit data movement, the shared-memory version may have a slightly lower performance. Our actual experimental results on the GTX480 GPU, however, show that the D-cache version, (i.e., the code in Figure 1b) is 43.8% slower than the shared-memory version (i.e., the code in Figure 1a).

To explain the unexpected experimental results, we first use a micro-benchmark, similar to the one used in [16], to measure the shared-memory access latency and the L1 D-cache hit latency. Our experimental results show that for GTX480, the shared-memory access latency is 44 cycles and the L1 D-cache

hit latency is 80 cycles. However, considering the fact that the benchmark matrix multiplication has a high degree of TLP to hide such latencies, the latency difference may not be a reason for our observed performance difference. Also, it is not clear how much the data movement overhead costs on the actual hardware. Therefore, we resort to a microarchitectural-level timing simulator, GPGPUsim V3.2.1 [2], to simulate the two code versions on a GPU configuration similar to GTX480. The shared-memory access latency and the L1 D-cache hit latency are 3 cycles and 1 cycle, respectively, in this simulator shared memory and the L1 D-cache both have the same throughput of 1 access per cycle. The results from the simulator show a trend similar to our actual hardware measurements, i.e., the D-cache version is 112.2% slower than the shared-memory version.

```
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
     AS(ty, tx) = A[a + WA * ty + tx];  //tx = threadIdx.x
     BS(ty, tx) = B[b + WB * ty + tx]; //tx = threadIdx.y
      __syncthreads();
  #pragma unroll
      for (int k = 0; k < BLOCK_SIZE; ++k)
           Csub += AS(ty, k) * BS(k, tx);
      __syncthreads();
  }
```

(a)

```
for (int a = aBegin, b = bBegin;  a <= aEnd; a += aStep, b += bStep)
{   #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
       Csub += A[a+WA*ty+k]*B[b+k*WB+tx];
            // tx = threadIdx.x and ty = threadIdx.y
    }
}
```

(b)

Figure 1. Code segments for the matrix multiplication kernel. (a) The kernel explicitly uses shared memory to store tiles (i.e., the shared-memory version); (b) the kernel implicitly uses the L1 D-cache to store tiles (i.e., the D-cache version).

We collected the dynamic instruction counts for both versions of code shown in Figure 1, indicating that the shared-memory version has 12.7% more instructions than the D-cache version. This confirms our initial analysis on the data movement overhead of the shared-memory version but offers no explanation why the shared-memory version has much higher performance.

Suspecting that the D-cache may suffer from conflict misses since the capacity is sufficient for the working sets, we use a perfect cache without any misses. Our results show that even with a perfect cache, the shared-memory version is still 0.2% faster than the D-cache version.

Puzzled with the results, we craft a microkernel to check whether it is possible for a D-cache version to outperform its equivalent shared-memory version. This microkernel loads a block of data and then re-accesses the same block many times. The results from this micro-benchmark confirm that the D-cache version is 13.0% faster than the corresponding shared-memory version (i.e., the block of data is located in shared memory). The data movement in the shared-memory version accounts for 8.3% dynamic instruction count overhead in this microkernel. With this result, now we are facing two key questions for the matrix multiplication kernel: (a) Why is the D-cache version slightly slower than the shared-memory version even with a perfect cache? (b) With a realistic cache, why is the D-cache version much slower?

To answer the first question, we look into the cycle-by-cycle instruction execution through the GPU pipeline. We observe that the load instruction corresponding to the array access '*A[a+WA*ty+k]*' in Figure 1b experiences additional pipeline stalls. The reason is that for a warp of 32 threads, when the TB dimension is 16x16, will have two rows of threads. For example, threads in warp 0 in TB 0 will have the thread id along the *X* direction, *threadIdx.x*, ranging from 0 to 15 and the thread id along the *Y* direction, *threadIdx.y* (i.e., *ty* in *A[a+WA*ty+k]*), ranging from 0 to 1. As a result, for this warp, the load instruction will access *A[a+WA*0+k]* and *A[a+WA*1+k]*. With a cache-line size of 128 bytes, these two accesses will fall into two separate cache lines. In other words, two cache accesses are needed to complete this load instruction for each warp using the D-cache version. In contrast, for the shared-memory version, the tiles are stored into shared memory before the loop. Then, inside the loop, the array access '*AS(ty,k)*' will correspond to *AS(0,k)* and *AS(1,k)* for warp 0. With shared memory featuring 32 banks and the dimension of the array *AS* of 16x16, both the elements *AS(0,k)* (i.e., AS[k]) and *AS(1,k)* (i.e., AS[1x16+k]) are located in the same row with *k* ranging from 0 to 16. Therefore, one shared memory access completes the shared-memory load instruction corresponding to '*AS(ty,k)*'. Since the array access '*A[a+WA*ty+k]*' of the D-cache version cannot be coalesced into a single cache access, it suffers from additional pipeline stalls even though all accesses hit in cache. Such overhead overweighs the benefit of fewer instruction counts compared to the shared-memory version, thereby resulting in lower performance even with a perfect cache.

To answer the second question, we first vary the cache associativity from the default value of 6 to fully associative. The resulting cache miss rate improves from 3.59 MPKI (misses per 1k instructions) to 1.79 MPKI, meaning that there are conflict misses due to limited set associativity. The performance is also improved by 12.8%. We also vary the capacity from 16kB to 48kB but the miss rate does not vary significantly. This is reasonable as the working set of an SM, i.e., 5TBs, is just 10kB. Therefore, considering the large performance gap, cache misses should not be the key culprit of the poor performance of the D-cache version. Again, we resort to cycle-by-cycle instruction-level analysis to understand what happens at run-time. We found that performance is not determined by the total number of cache misses. Instead it depends more on how these cache-misses overlap with each other, i.e., the degrees of memory-level parallelism (MLP).

With the D-cache version, for each warp, during each iteration of the inner loop '*for (int k = 0; k < BLOCK_SIZE; ++k)*', the access to array B '*B[b+k*WB+tx]*' is a cache miss (e.g., *tx* = 0~15 for warp 0, warp 1, etc.) due to the large value of '*WB*'. In other words, for each different *k*, the access '*B[b+k*WB+tx]*' is a cache miss. The access to array A '*A[a+WA*ty+k]*' results in two misses (e.g., *ty*=0~1 for warp 0, ty = 2~3 for warp 1, etc.) for the first iteration (i.e., *k*=0) and

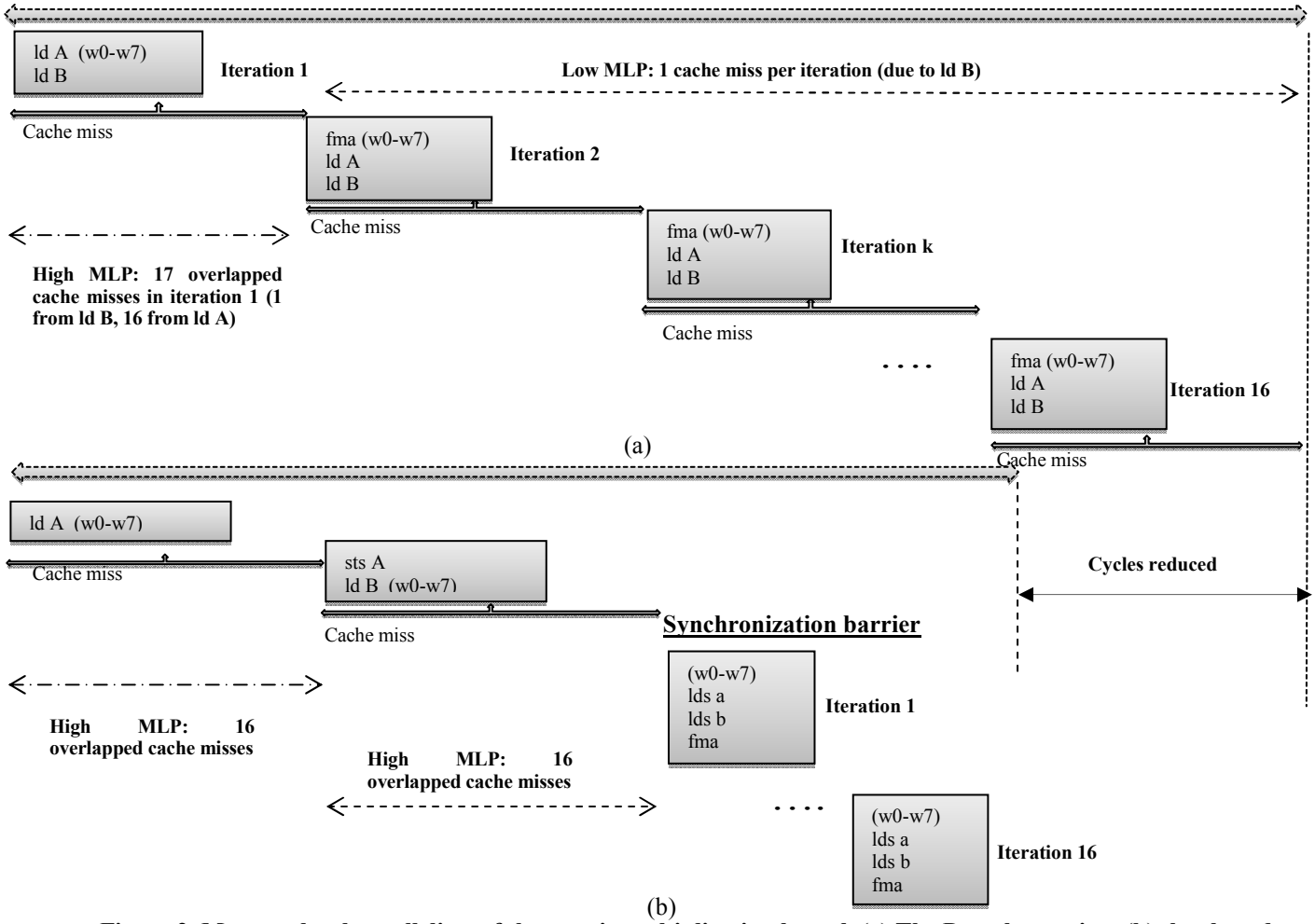

Figure 2. Memory-level parallelism of the matrix multiplication kernel. (a) The D-cache version, (b) the shared-memory version.

hits in the cache for subsequent iterations. These two loads are followed by a dependent floating-point multiply-and-add (*fma*) instruction. Therefore, the cache misses are distributed across iterations. Considering a TB with a round-robin warp scheduling policy, the warps make similar progress and different warps can also overlap their misses, as shown in Figure 2a. However, as the access to array B '*B[b+k*WB+tx]*' has an index independent on the thread id along the *Y* direction, with a TB size of 16x16, all warps in a TB will access the same address during the k^{th} loop iteration. In other words, the misses due to accessing array B remain distributed across loop iterations. In comparison, for the shared-memory version, the code before the inner loop forces the tiles to be loaded from global memory and stored in shared memory before computation. For array B, the access '*B[b+WB*ty+tx]*' ensures that the entire tile from array B will be loaded by different warps: warp 0 accesses '*B[b+WB*ty+tx]*' with *tx*=0~15 and *ty*=0~1; warp 1 accesses '*B[b+WB*ty+tx]*' with *tx*=0~15 and *ty*=2~3, etc. Therefore, all the cache misses from different warps in a TB are aggregated and overlapped with each other, as shown in Figure 2b. Comparing Figure 2a with Figure 2b, we can see that much higher MLP is achieved with the shared-memory version. Therefore, with a realistic cache, the performance of shared memory is much better than the D-cache version although both versions have similar numbers of total cache misses.

```
for (int a = aBegin, b = bBegin;  a <= aEnd; a += aStep, b += bStep)
{
    prefetch(A[a + WA * ty + tx]);  //Prefetch the tile of matrix A
    prefetch(B[b + WB * ty + tx]);  //Prefetch the tile of matrix B
    __syncthreads();
    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub += A[a+WA*ty+k]*B[b+k*WB+tx];
                // tx = threadIdx.x and ty = threadIdx.y
    }
}
```

Figure 3. The D-cache version of matrix multiplication with prefetching instructions to improve MLP.

One way to improve MLP for the D-cache version is to use prefetch instructions to aggregate the cache misses together, as shown in Figure 3. However, comparing the code shown in Figure 1a and Figure 3, we can see that they have similar complexity and since the subtle MLP behavior is not obvious, it would be non-intuitive to engage in such a prefetching optimization. Furthermore, even with prefetching instructions, the code in Figure 3 still has a lower performance (8%) than

the shared-memory version in Figure 1a due to non-coalesced cache accesses.

In summary, for matrix multiplication, the two key reasons for its shared-memory version to outperform the D-cache version are (a) its high MLP as the tiles are loaded into shared memory before computation; and (b) shared memory has 32 banks and is much less susceptible to the coalescing problem compared to the D-cache. In comparison, the classical metric of cache performance, i.e., the cache-miss/hit rate, is less critical than these two factors.

B. Case Study II: Fast Fourier Transform (FFT)

In this case study, we use a radix-4 based 1-D FFT kernel. The test input is a 1k-point FFT with a batch size of 4k. For a 1k-point FFT, each thread goes through multiple stages with each stage performing a 4-point FFT and then exchanging data with other threads. As a result, the key to achieve high throughput is to hold all the intermediate results in shared memory or the D-cache. The sample code of the radix-4 based FFT using shared memory and the D-cache is shown in Figure 4. Comparing the two versions, we can see that they are identical except that one uses a shared-memory array to keep the intermediate results and the other uses a global-memory array to do so. If the cache is large enough to keep the temporary data, all the accesses will be cache hits and enjoy low access latency.

For the two FFT kernels, we first test them on a GTX480 GPU. For the shared-memory version, the shared-memory capacity (48kB) limits the number of concurrent TBs on an SM to be 5. For fair comparison, we also apply the same limit to the D-cache version with an unused share-memory array in the code. We also vary the TB parameter in our experiments in Section IV. Our experimental results show that the shared-memory version is 3.77X faster than the D-cache version.

The first reason for this high performance disparity is that on the GTX480, the L1 cache adopts a write-evict (WE) policy. As seen from Figure 4b, after each 4-point FFT computation, the results are written to the global-memory array, meaning that the data will be evicted from the cache. Therefore, the L1 cache provides no benefits for subsequent reads. To further analyze the impact of the cache-write policy, the microarchitectural-level simulator, GPGPUsim, is used again. With a WE policy, it reports that the D-cache version is 119% slower than the shared-memory version. When we change the write policy to write-back and write-allocate (WBWA), the performance of the D-cache version improves by 26% but is still much slower than the shared-memory version.

To determine the reason, we then change the L1 D-cache to be fully associative. Compared to the default 6-way set associative configuration, both the total number of cache misses and the overall execution time changes very little (less than 1%). We also increase the cache capacity to 64kB, 128 kB, and 1MB and there is little performance impact. Therefore, cache contention and cache capacity are not the reason why the D-cache version under-performs the shared-memory version. Also, unlike matrix multiplication, where the tiles are loaded to shared memory prior to computation, for FFT, both the shared-memory version and the D-cache version load the global-memory data and access the intermediate results in the same way, as shown in Figure 4. So, there is not much difference in their MLP either.

```
loadFromGlobal();
FFT4(0);
saveToSM(0); //define by multiple threads in a TB
__syncthreads();
loadFromSM(0); //use by multiple threads in a TB
FFT4(1);
__syncthreads();
saveToSM(1); //(re)define by multiple threads in a TB
__syncthreads();
loadFromSM(1); //use by multiple threads in a TB
....
FFT4(4);
writeToGlobal();
```

(a)

```
loadFromGlobal();
FFT4(0);
SaveToGlobal(0); //define by multiple threads in a TB
__syncthreads();
LoadFromGlobal(0); //use by multiple threads in a TB
FFT4(1);
__syncthreads();
SaveToGlobal(1); //(re)define by multiple threads in a TB
__syncthreads();
LoadFromGlobal(1); //use by multiple threads in a TB
....
FFT4(4);
writeToGlobal();
```

(b)

Figure 4. The code for 1-D FFT. (a) The shared-memory version; (b) the D-cache version.

Compared to shared memory, the L1 D-cache with a write-allocate policy may suffer a penalty due to write misses, which is the case for FFT when the intermediate result array is updated after the first stage of a 4-point FFT. If only a part of a cache line is to be updated, the full cache line will need to be fetched first from the off-chip memory, incurring a high penalty. To analyze such impact, we change the simulator such that the first write misses to the intermediate result array are treated as hits. With such a change, the performance of the D-cache version is improved by 21%.

Another key factor that affects the performance of the D-cache version is the un-coalesced cache accesses. As shown in Figure 5, the 1-D shared memory array 'sh_mem' is used for intermediate results for the shared-memory version. During one 4-point FFT pass of the 1k-point FFT computation, the intermediate results are stored to the 'sh_mem' array using the function 'SavetoSm'. The array index or offset is computed with '(tid<<2)+(tid>>3)' where 'tid' is generated from the thread ids as shown in Figure 5. Such an index computation eliminates the bank conflicts in shared memory accesses and each shared-memory load instruction from a warp can be satisfied with a single shared-memory read transaction. For example, for warp 0 in TB0, its 'tid' ranges from 0 to 31. With '(tid<<2)+(tid>>3)', the mapped indices become 0, 4, 8, 12, 16, 20, 24, 28, 33, 37, 41, 45, etc. Considering that there are 32 banks in shared memory, different threads in a warp will access a different bank in shared memory. However, when we replace this shared-memory array with a global memory array, the L1 D-cache cannot leverage such interleaved indices. These indices cannot be coalesced to a single cache line access and

instead have to be mapped to 4 different cache lines with a cache line size of 128 bytes. As a result, each array update (i.e., a store-to-global-memory instruction with such indices/offsets) will require 4 updates to the L1 D-cache. The same access patterns are also present in the load operations. Considering the L1 D-cache/shared memory has one read port and one write port, each L1 D-cache access incurs significantly higher latency. Therefore, the D-cache version achieves much lower performance than the shared-memory version.

```
#define   SavetoSm(distance, stide_xyzw, offset) { \
  float *pv_shm=sh_mem+offset; \
  For(int i=0;i<4;i++)   { \
    pv_shm[0*stride_xyzw] = zr[i].x; \
    pv_shm[1*stride_xyzw] = zr[i].y; \
    pv_shm[2*stride_xyzw] = zr[i].z; \
    pv_shm[3*stride_xyzw] = zr[i].w; \
    Pv_shm+=distance;  \
 } \}
__global__ void fft1k(float * greal, float * gimg) {
  __shared__ float sh_mem[68*4*4*2];
  uint gid = threadIdx.x+blockIdx.x*blockDim.x;
  uint tid = gid & 0x3fU;
  ...
  FFT4()
  SavetoSm(66*4, 1, （tid<<2)+(tid>>3));
  __syncthreads();
  ...}
```

Figure 5. The code for accessing the intermediate result array (sh_mem) in the benchmark FFT.

In summary, for the benchmark FFT, three key factors affect the performance of the D-cache version. First, the write-evict policy makes the cache useless as there are updates to the intermediate result array before it is read again. Second, even with a write-back write-allocate L1 D-cache, the first write misses incur non-trivial performance penalty. Third, the un-coalesced cache accesses significantly increase the latency of each load/store from/to the intermediate result array.

C. Case Study III: Matching Cube (MC)

The benchmark MC has multiple kernels and the kernel '*Generatetriangles*' dominates the performance. For this kernel, each TB has 32 threads and the launched grid contains 1024 TBs. The code segment of this kernel is shown in Figure 6a. It uses two shared memory arrays, '*vertlist*' and '*normlist*' for its intermediate results. The indices used to access these two arrays, as shown in Figure 6a, are '*threadIdx.x + i*NTHREADS*', where '*NTHREADS*' is 32 and '*i*' ranges from 0 to 11. As a result, the data stored in these two shared memory arrays are actually not shared among different threads in a TB. Instead, each thread uses a part of each array for its own intermediate data. Therefore, we can replace these two shared-memory arrays with local-memory arrays, which are private to each thread, and use the L1 D-cache to buffer such local memory arrays. The code utilizing the local-memory arrays is shown in Figure 6b. The main reason why the local-memory arrays are used rather than global-memory arrays is that the L1 D-cache in the Fermi architecture uses a write-back write-not-allocate policy for local memory data.

Given the size of the two shared memory arrays (2*4*3*12*32 = 9216 Bytes) in the shared-memory version, each SM can run 5 TBs concurrently when its shared memory size is configured to 48kB. For the D-cache version, as it does not use shared memory at all, each SM can run up to 8 TBs concurrently, where 8 is the maximal number of TBs to run on an SM. We test the two versions of the MC code on the GTX480 GPU. The results show that the D-cache version has 17.3% higher performance than the shared-memory version due to the higher number of concurrent TBs. If we use a fake shared memory array to set the number of concurrent TBs for the D-cache version to be 5, the D-cache version is 5.8% slower than the shared-memory version due to the write-back write-not-allocate policy. Our experiments on GPGPUsim also report similar performance trends.

```
__global__ void generateTriangles(…) {
   …
   __shared__ float3 vertlist[12*NTHREADS]; //NTHREADS = 32
   __shared__ float3 normlist[12*NTHREADS];
   //defines to the shared memory array
   vertexInterp2(isoValue, v[0], v[1], field[0], field[1],
vertlist[threadIdx.x], normlist[threadIdx.x]);
   vertexInterp2(isoValue, v[1], v[2], field[1], field[2], vertlist[threadIdx.x+NTHREADS],
normlist[threadIdx.x+NTHREADS]);
   vertexInterp2(isoValue, v[2], v[3], field[2], field[3],
vertlist[threadIdx.x+(NTHREADS*2)], normlist[threadIdx.x+(NTHREADS*2)]);
   …
   //uses of the shared memory array
   pos[index] = make_float4(vertlist[(edge*NTHREADS)+threadIdx.x], 1.0f);
   …
}
```

(a)

```
__global__ void generateTriangles(…) {
   …
   float3 vertlist[12];
   float3 normlist[12];
   ///defines to the local memory array
   vertexInterp2(isoValue, v[0], v[1], field[0], field[1], vertlist[0], normlist[0]);
   vertexInterp2(isoValue, v[1], v[2], field[1], field[2], vertlist[1], normlist[1]);
   vertexInterp2(isoValue, v[2], v[3], field[2], field[3], vertlist[2], normlist[2]);
   …
   //uses of the shared memory array
   pos[index] = make_float4(vertlist[edge], 1.0f);
   …
}
```

(b)

Figure 6. The code segment of the benchmark Matching Cube (MC). (a) The shared-memory version, (b) the D-cache version.

In summary, for the benchmark MC, the non-sharing usage of its shared-memory arrays enables them to be replaced with local-memory arrays. The key reason why the D-cache version outperforms the shared-memory version is that the D-cache version eliminates the shared-memory usage, which in turn removes the resource limitation on the number of concurrent TBs on each SM. Such improved TLP leads to higher performance for MC.

D. Case Study IV: Path Finder (PF)

The kernel code of the benchmark Path-Finder makes use of two shared memory arrays, '*prev*' and '*result*', as shown in Figure 7a. Its TB dimension is 256x1 and its thread grid size is 19x1. As a result, the sizes of these two shared-memory arrays are small (256x4=1kB) and such shared memory usage is not a bottleneck for the number of concurrent TBs on each SM. For the shared-memory array '*prev*', its accesses in the kernel code, '*prev[tx-1]*' and '*prev[tx+1]*'indicate that the data in this array are shared among different threads. In contrast, the shared-memory array 'result' is always accessed with '*result[tx]*', which means that there is no data sharing for this array. Therefore, we can simply use a register variable to replace it, as shown in Figure 7b. Further, as the variable is only defined and used in the same thread, we can safely remove the synchronization instruction '__syncthread()' after the statement

defining the variable '*result*', as shown in Figure 7b. The same argument can also be made to remove the '__synchthread()' instructions after defining the '*result*' array element, as shown in Figure 7a.

```
__global__ void dynproc_kernel(…){
   __shared__ float prev[BLOCK_SIZE];
   __shared__ float result [BLOCK_SIZE];
   int tx=threadIdx.x ;
   …
   if(IN_RANGE(xidx, 0, cols-1))
      prev[tx] = gpuSrc[xidx];
   …
   for (int i=0; i<iteration ; i++){
     if( IN_RANGE(tx, i+1, BLOCK_SIZE-i-2)){
        shortest = minum( prev[tx-1], prev[tx],prev[tx+1]);
        result[tx] = shortest + gpuWall[index];
        __syncthreads();  //Can be safely removed
        prev[tx]= result[tx];
        __syncthreads();
  }
gpuResults[xidx]=result[tx]; }
```

(a)

```
__global__ void dynproc_kernel(…){
   __shared__ float prev[BLOCK_SIZE];
   float result;
   …
   for (int i=0; i<iteration ; i++){
     if( IN_RANGE(tx, i+1, BLOCK_SIZE-i-2)){
        shortest = minum( prev[tx-1], prev[tx],prev[tx+1]);
        result = shortest + gpuWall[index];
        //__syncthreads();   //removed as there is no race on result.
        prev[tx]= result;
        __syncthreads();
   }
gpuResults[xidx]=result;}
```

(b)

Figure 7. The code segment of the benchmark Path-Finder (PF). (a) The shared-memory version, (b) the D-cache version.

As register accesses have the lowest latency and the shared-memory accesses also involve additional instructions to compute the addresses, the D-cache version shown in Figure 7b outperforms the shared-memory version shown in Figure 7a by 11.2%. If we do not remove the the '__syncthread()' instruction from the D-cache version in Figure 7b, it still outperforms the shared-memory version by 6.5%, highlighting the performance benefits by using registers to replace the shared-memory array '*result*'. In comparison, if we remove the '__syncthread()' instruction from the shared-memory version, its performance improves by 4.7%, which still has lower performance than the D-cache versions. Using GPGPUsim, we find that by using registers to replace the shared-memory array, the D-cache version has 3.6% fewer dynamic instructions than the shared-memory version. When the '__syncthreads()' instruction is removed, the D-cache version has 5.4% fewer instructions than the shared-memory version. However, if we replace the prev array in shared memory with a global-memory array, the performance would degrade by 15.7%. . This is mainly because re-accessing the frequently updated prev array in each loop hurts the cache performance with the WE policy.

In summary, for the benchmark PF, as one of its shared-memory array is not used for data communication across different threads, this shared-memory array can be replaced with register variables. The register variables have the lowest access latency and it also takes fewer instructions to access registers than accessing shared-memory variables. Both lower latency and fewer dynamic instructions make the D-cache version outperform the shared-memory version for PF.

IV. EXPERIMENTS

A. Experimental Methodology

In our experiments, we use both real GPU hardware and a microarchitectural timing simulator, GPGPUsim V3.2.1. Our experiments on the GTX480 GPU are performed on a Linux workstation with CUDA 4.0. We also perform experiments on a GTX680 GPU with CUDA 5.0. In our experiments using GPGPUsim V3.2.1, we use the configuration shown in Table 1, which is similar to the Fermi architecture, i.e., GTX 480 GPUs. As the L1 D-cache and shared memory utilize the same 64kB storage, we use 48kB L1 D-cache + 16kB shared memory for the L1 D-cache versions and 16kB L1 D-cache + 48kB shared memory for the shared-memory versions.

Table 1. The GPGPUsim configuration.

# of execution cores	15
SIMD Pipeline Width	16
Number of Threads/Core	1536
Number of Registers/Core	32768
Shared Memory /Core	16kB/48kB: 32 banks; 3-cycle latency; 1 access per cycle
L1 Data cache/Core	16kB: 128B line, 4-way assoc 48kB: 128B line, 6-way assoc 1-cycle hit latency
MSHR Entry	128 entries
Constant Cache Size /Core	8k
Texture Cache Size/Core	12k , 128B line, 24-way assoc
L2 Data cache	768k: 128B line, 16-way assoc
Number of Memory Channels	6
Memory Channel Bandwidth	8 Bytes/Cycle
DRAM clock	1400 MHz
DRAM Schedule Queue Size	16 , Out of Order (FR-RCFS)
Warp Scheduling Policy	Greedy then oldest scheduler

In this study, in order to cover the most common ways to utilize shared memory, we select 15 benchmarks that have shared memory usage from the NVIDIA CUDA SDK [12], AMD OPENCL SDK [1], GPGPUsim suite [2], Rodinia [4], and Opencv [1]. We also include a tiled version of matrix-vector multiplication [18] with a tile size of 32x32. The input and the kernel invocation parameters of these applications are shown in Table 2. Based on the way that shared memory data are reused, we classify the benchmarks into two categories. The first category uses shared memory for inter-thread communication and its data reuses are across threads. For applications in this category, we use global memory arrays to replace shared memory arrays, if needed. In the second category, applications use shared memory for intra-thread data reuse and we use local memory arrays or registers to replace shared memory variables. The second category includes the benchmarks STO, MV, MC, NQU, and PF. The remaining applications fall into the first category.

For a benchmark, if its shared-memory version limits the number of concurrent TBs on each SM, its corresponding D-cache version will have no such a constraint. In such cases, we allocate an unused shared-memory array to control the number of concurrent TBs to run on each SM. Then, we pick the best performing TB number for the D-cache version of the

benchmark. More details on the TB number are discussed in Section IV-D.

Table 2. The benchmarks used in experiments.

Name	Threads	Blocks	Input size
Matrix Multiplication(MM) [12]	(16,16)	(16,16)	256*256
Histogram(HG) [1]	(64,1)	(128,1)	1024*1024
Matrix Transpose(MT) [12]	(16,16)	(64,64)	1024x1024
Fast Walsh Transform(FWT) [12]	(512,1)	(4096,1)	8388608
3D Laplace Solver(LPS) [2]	(32,4)	(4,25)	100*100*100
Needleman-Wunsch(NW) [4]	(16,1)	(64,1)	1024*1024*10
Back Propagation(BP) [4]	(16,16)	(16,16)	65536
scalar product of vectors(SP) [12]	(64,1)	(256,1)	8192*512
Fast Fourier Transform(FFT) [1]	(64,1)	(4096,1)	4k*1k
Discrete Wavelet Transform(DWT) [12]	(512,1)	(256,1)	256k
Image Blur(Blur) [1]	(256,1)	(2,512)	1k*1k
StoreGPU(STO) [2]	(128,1)	(384,1)	196656
Matrix-Vector Multiplication(MV)[18]	(32,1)	(64,1)	2048*2048
MarchingCubes(MC) [12]	(32,1)	(1024,1)	32768
N-Queue Solver(NQU) [2]	(96,1)	(256,1)	8
Path Finder(PF) [4]	(256,1)	(19,1)	4096*100*20

B. Experimental Results on GTX 480 and GTX680 GPUs

We first present a performance comparison between the shared-memory versions and the D-cache versions on the GTX 480 GPU. As shown in Figure 8, the execution time of the D-cache version of each benchmark is normalized to the execution time of its corresponding shared-memory version. From the figure, we can see that on GTX 480, among the 16 benchmarks in our study, 14 benchmarks favor shared memory. Only two benchmarks, PF and MC, have higher performance with L1 D-cache versions than their shared-memory versions for the reasons discussed in Section III. On average, using the geometric mean (GM), the shared-memory versions result in 55.7% higher performance than the L1 D-cache versions.

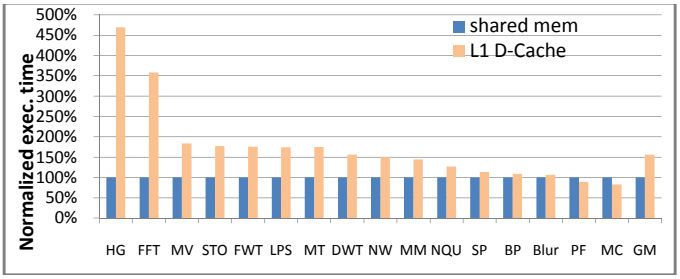

Figure 8. Performance comparison between the shared-memory versions and the L1 D-cache versions on a GTX 480 GPU.

A performance comparison for the GTX 680 is presented in Figure 9. Compared to the Fermi architecture (i.e., GTX480), the Kepler architecture (i.e., GTX680) has more ALUs in each SM. Also, the L1 D-cache/shared memory in the Kepler architecture has a lower hit latency (33/33 cycles) than in the Fermi architecture (80/44 cycles). Unlike the Fermi architecture, global memory data are not cached in the L1 D-cache in the Kepler architecture. Despite these architectural differences, among the 16 benchmarks, only these same two benchmarks, PF and MC, achieve higher performance using the L1 D-cache version than the shared memory version.

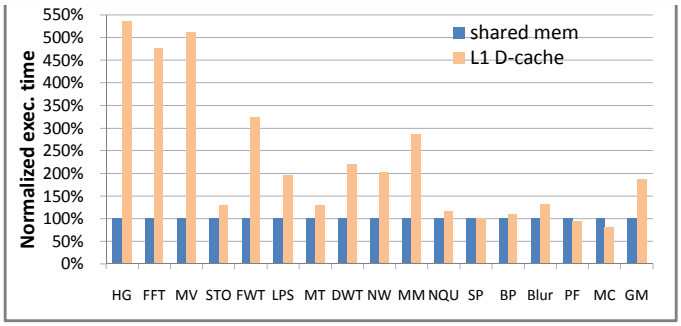

Figure 9. Performance comparison between the shared-memory versions and the L1 D-cache versions on a GTX 680 GPU.

C. Experimental Results on GPGPUsim

The performance comparison between the shared-memory versions and the L1 D-cache versions on GPGPUsim is presented in Figure 10. The performance for the L1 D-cache versions using both a write-evict (WE) policy and a write-back write-allocate (WBWA) policy are normalized to the performance of the shared memory versions. We also include the results of a fully associative L1 D-cache (FA+WBWA).

Compared to the actual hardware, the L1 D-cache has a 1-cycle hit latency and shared memory has a 3-cycle access latency in our GPGPUsim model. As a result, some benchmarks, such as FFT, NQU, and Blur, show smaller performance gaps between the L1 D-cache versions and the shared memory versions than the GTX480/680 results. On the other hand, for the benchmark MV, GPGPUsim reports much higher performance gaps between the L1 D-cache versions and the shared memory versions than the GTX480/680 results. The reason is that the L1 D-cache in GPGPUsim uses a basic index function, which leads to high numbers of conflict misses for MV.

Several interesting observations can be made from Figure 10. First, for most benchmarks, the shared memory versions have higher performance than the D-cache versions, which is consistent with the real hardware results shown in Figure 8 and Figure 9.

Second, the WE policy hurts the cache performance if the updated data will be re-accessed again soon. This is the case for the benchmark FFT, as discussed in Section III-B. The benchmarks HG, MV, FWT, LPS, DWT, and BP also show similar behavior. Therefore, a WBWA policy can significantly improve the performance for these benchmarks. However, if

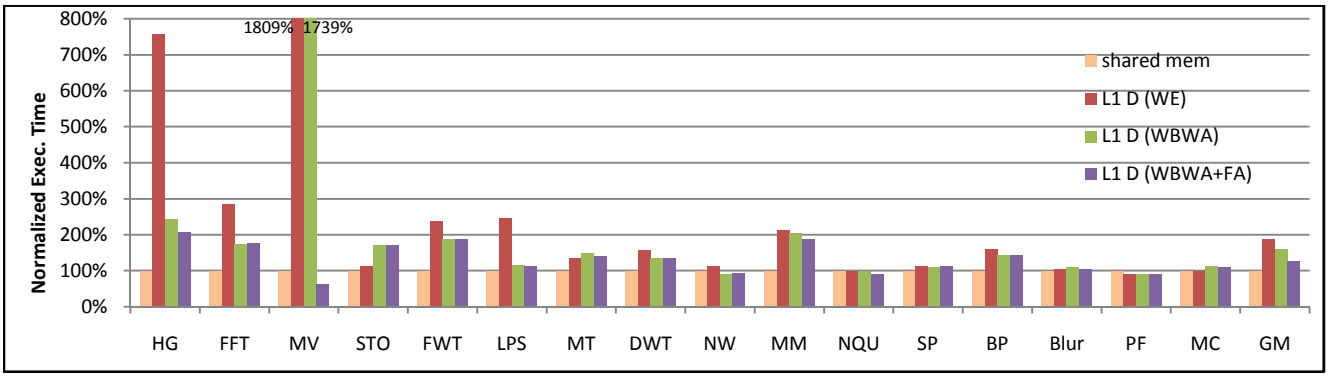

Figure 10. Performance comparison between the shared-memory versions and the L1 D-cache versions on GPGPUsim. The following cache models are evaluated for D-cache versions: 6-way 48kB with a write evict (WE) policy, 6-way 48kB with a write-back write-allocate (WBWA) policy, fully-associative 48kB with a WBWA policy (WBWA+FA).

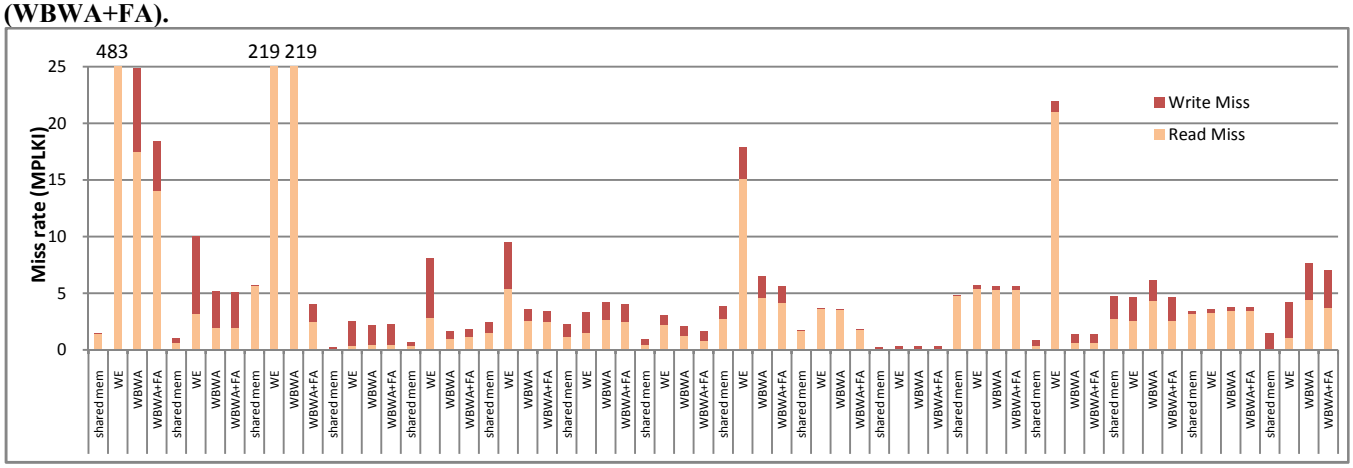

Figure 11. Cache miss rates of the shared-memory versions and the L1 D-cache versions on GPGPUsim. The following cache models are evaluated for D-cache versions: 6-way 48kB with a write evict (WE) policy, 6-way 48kB with a write-back write-allocate (WBWA) policy, fully-associative 48kB with a WBWA policy (WBWA+FA). The Shared Memory versions use a 4-way 16kB L1 D-cache.

the updated data will not be re-accessed, the WBWA policy may introduce cache pollution, thereby increasing the cache miss rate. The benchmarks STO, MT, and MC exhibit such a problem with the WBWA policy, as confirmed in the cache miss results presented in Figure 11.

Third, increasing set associativity reduces conflict misses and the benchmark MV drastically benefits from it. With the tiling optimization, the working set size of MV is within the cache capacity. However, its threads in a warp access the input matrix in a column-by-column manner. Therefore, given an input-matrix size of 2048x2048, different columns have the same cache line indices, resulting in a high number of conflict misses. A fully associative cache eliminates the problem, as also confirmed in Figure 11.

Fourth, with a WE policy, the D-cache versions outperform the shared-memory versions only for the benchmarks PF and MC, which is also consistent with the actual hardware results. With a fully associative cache and a WBWA policy, the benchmarks MV, NQU, and PF show higher performance using the D-cache versions than the shared-memory versions. For MV, since there is no data communication among the threads, we use registers to buffer each tile of the input matrix. As discussed before, register accesses have a lower latency and require a smaller number of instructions than shared-memory accesses. Furthermore, using registers eliminates the need for coalescing to perform column-based accesses. Therefore, the D-cache version using a fully associative cache for MV outperforms the corresponding shared-memory version. As discussed in Section III-C, the D-cache version for the benchmark MC benefits from increased TLP, thereby resulting in higher performance than its shared-memory version. The benchmark NQU also shows similar behavior to MC.

In Figure 11, we compare the cache performance, measured as misses per kilo instructions (MPKI), between the shared-memory versions and the D-cache versions. Note that the shared-memory versions still utilize the L1 D-cache when they access global memory or local memory. Also, for the shared-memory versions, the L1 D-cache size is 16kB. From Figure 11, we can see that the shared-memory versions typically have low cache miss rates, indicating that most data accesses are satisfied with shared memory. For the D-cache versions, a WBWA policy reduces the misses caused by write-evictions but may hurt some benchmarks as discussed before. From

Figure 10 and Figure 11, we can also see that TLP significantly mitigates the performance impact of cache misses.

D. The TLP Impact of the D-Cache Versions

Among the benchmarks in our study, the shared-memory usage for the benchmarks FFT, MC, NQU, HG, and STO imposes a limitation on the number of TBs that can run concurrently on an SM. When the shared-memory usage is replaced, such a limit is eliminated and more TBs can be dispatched to an SM. We explore the number of TBs for the D-cache versions and the performance results measured on the GTX 480 GPU are shown in Figure 12. The results on the GTX680 GPU and GPGPUsim show similar trends. Among these benchmarks, the maximal number of concurrent TBs per SM is 4 for the benchmark STO due to its register usage. For the remaining four benchmarks, up to 8 TBs can be dispatched to an SM, where 8 is a hardware limit [10]. From Figure 12, we can see that the benchmarks MC and STO prefer more TBs or higher TLP. For NQU, the TLP impact is relatively small as long as there are at least 3 TBs in each SM. For benchmarks FFT and HG, the best performance is achieved with 4 TBs and 6 TBs per SM, respectively. Further increasing the number of TBs per SM will result in cache contention, thereby hurting performance.

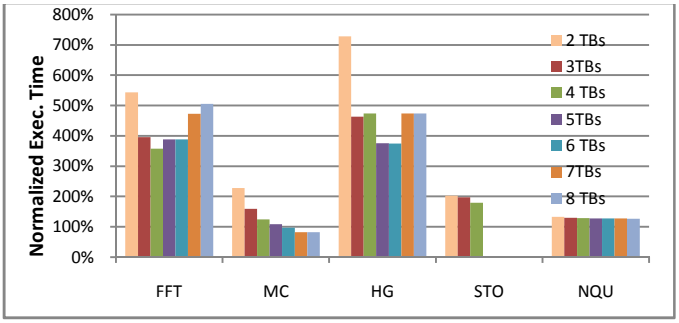

Figure 12. The execution time of D-cache versions with different numbers of concurrent TBs per each SM. The execution times are normalized to the execution time of the shared-memory versions.

E. Impact on Energy Consumption

We use GPUWattch[9] to analyze the impact of using shared memory vs. L1 D-cache on energy consumption and the results are in Figure 13. Similar to our performance comparison, we include the energy consumption results for L1 D-cache versions using a write-evict (WE) policy, a write-back write-allocate (WBWA) policy, and a fully associative write-back write allocate L1 D-cache (FA+WBWA). The energy of D-cache versions is normalized to the energy of the shared memory versions.

As shown in Figure 13, the trend on energy matches the performance trend shown in Figure 10 for most benchmarks due to the significant static energy consumption modeled in GPUWattch and the lack of speculative execution in GPUs. One interesting exception is the D-cache version for the benchmark MC. The D-cache version enables more concurrent thread blocks (TB) and has lower execution time than its shared memory version. However, the high number of TBs leads to cache contention and results in higher dynamic energy consumption in memory hierarchy. Therefore, for MC, the D-cache version, although with lower execution time, consumes more energy than the shared-memory version.

Overall, for most of the benchmarks, the shared memory version consumes less energy than the D-cache version. On average, the shared memory versions consume 48.5% energy compared to the D-cache versions with a WE policy, 53.7% energy compared to the D-cache versions with a WBWA policy, and 71.9% of energy compared to the D-cache versions with a FA and a WBWA policy.

F. Summary

In summary, from our experiments on both real hardware and architectural simulators, we can derive the following important insights. First, MLP and coalesced accesses are key reasons for the shared-memory versions to outperform the D-cache versions. Second, for D-cache versions, the D-cache write policy and D-cache contention may have a strong impact on performance. Third, eliminating shared-memory usage enables higher TLP and more opportunities to allocate variables to registers, which are the key reasons for the D-cache versions to outperform the shared-memory versions.

V. Related Work.

To the best of our knowledge, our work is the first to study the tradeoffs between software-managed cache and hardware-managed cache on GPU architectures. The most related work is by Jia et al. [8], who observed that the L1 D-caches are not always helpful for GPGPU performance. They identified as a reason that data fetched at the cache-line granularity may consume too much bandwidth if data reuses are limited. They also propose a compiler-time approach to predict the D-cache's performance impact and then to turn on/off the D-cache accordingly. In comparison, we focus on comparing the effectiveness between the D-cache and shared memory. For our benchmarks, the D-cache versions are developed directly from the shared memory versions such that the cached data have strong data reuse, which lead us to identify different performance tradeoffs.

Software managed caches have been widely used in different architectures. In embedded systems, they are referred to as scratchpad memory and have been well studied [13][15]. In [3], Banakar et al. performed a comparison between scratchpads and caches for embedded systems and showed that scratchpads are a promising alternative to caches. In the CellBE architecture, local stores are essentially software managed caches and there have been numerous works on how to better utilize them for performance enhancement [6][7]. Compared to these architectures, GPUs have a high emphasis on TLP and are throughput oriented rather than latency oriented. The single-instruction multiple-thread (SIMT) style execution presents quite different performance tradeoffs between software-managed and hardware-managed caches. As shown in our experimental analysis, TLP, MLP, and coalescing have a strong performance impact while such factors are non-existent in embedded systems.

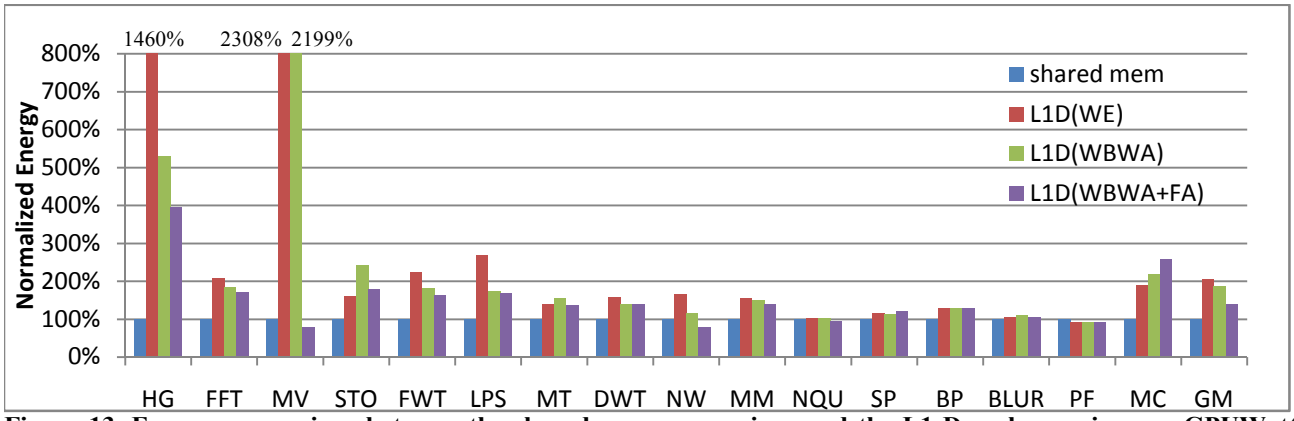

Figure 13. Energy comparison between the shared-memory versions and the L1 D-cache versions on GPUWattch. The following cache models are evaluated for D-cache versions: 6-way 48kB with a write evict (WE) policy, 6-way 48kB with a write-back write-allocate (WBWA) policy, fully-associative 48kB with a WBWA policy (WBWA+FA).

Current works on GPU memory hierarchy mainly focuses on how to improve performance. There have been approaches proposed to utilize shared memory to convert un-coalesced accesses into coalesced ones [17]. The TLP limit introduced by shared memory usage is addressed by timing-multiplexing shared memory [18]. Cache contention among warps has been observed in [14] and a cache-conscious warp scheduling algorithm is proposed to address this problem.

VI. CONCLUSIONS

In this paper, we present an in-depth study to reveal interesting and somewhat unexpected tradeoffs between shared memory and L1 D-caches in GPGPU applications. We generate the code to leverage D-caches from the code utilizing shared memory so as to ensure that the same optimization techniques such as tiling are applied equally. Since TLP can significantly hide the performance impact of L1 cache misses, many subtle factors can have more profound performance impacts than cache miss rates. Through detailed case studies, we show that the key reasons for shared-memory versions to outperform D-cache versions are MLP and coalescing (i.e., bank-conflict-free accesses). The D-cache versions mainly benefit from improved TLP and using registers to store variables. On the other hand, the D-cache versions may be affected by un-coalesced cache accesses, cache-write policies and cache contention. Among the 16 benchmarks in our study, the shared memory versions outperform the D-cache versions and consumes less energy for most of them, justifying the software complexity to manage shared memory even in the existence of D-caches. Such results hold irrespective of latency differences between the L1-D cache and shared memory, as shown by simulations and actual hardware experiments. On the other hand, for data private to each thread, storing them into registers or local memory to leverage L1 D-caches can be a better alternative to shared memory.

VII. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments to improve our paper. This work is supported by an NSF project 1216569, an NSF CAREER award CCF-0968667, a subcontract under AFOSR-FA9550-12-1-0442 and a grant from the DARPA PERFECT program.

REFERENCES

[1] AMD Accelerated Parallel Processing SDK http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/

[2] A. Bakhoda, et al., Analyzing CUDA workloads using a detailed GPU simulator. *ISPASS*, 2009

[3] R. Banakar, et al., Comparison of Cache- and ScratchPad-based Memory Systems with respect to Perfor,ance, Area, and Energy Consumption, Dekanat Informatik, Univ., Technical Report, 2001

[4] S. Che, et al., "Rodinia: A Benchmark Suite for Hetero-geneous Computing," *IISWC*, 2009

[5] CUDA programming guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[6] A. E. Eichenberger et al., "Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture," *IBM Sytems Journal*, 2006.

[7] M. Gonzalez et al., Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture, *PACT*, 2008

[8] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and improving the use of demand-fetched caches in GPUs, *ICS*, 2012.

[9] J. Leng, et al.. GPUWattch: Enabling Energy Optimization in GPGPUs. *ISCA,* 2013.

[10] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture, Nov. 2011

[11] NVIDIA Kepler GK110 white paper. 2012.

[12] NVIDIA. CUDA C/C++ SDK Code Samples, 2011. http://developer.nvidia.com/gpu-computing-sdk, 2011.

[13] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications, *EDTC*, 1997.

[14] T. Rogers, M. O'Connor, and T. Aamodt. "Cache-Conscious Wavefront Scheduling," MICRO, 2012

[15] S. Udayakumaran et al., Dynamic allocation for scratch-pad memory using compile-time decisions, *ACM TECS*, 2006.

[16] H. Wong, et al., Demystifying GPU microarchitecture through microbenchmarking, *ISPASS,* 2010.

[17] Y. Yang, P. Xiang, J. Kong and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. *PLDI*, 2010

[18] Y. Yang, et al., Shared Memory Multiplexing: A Novel Way to Improve GPGPU Performance. *PACT*, 2012.