Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput

Yi Yang
North Carolina State University
Raleigh, NC
yyang14@ncsu.edu

Ping Xiang
North Carolina State University
Raleigh, NC
pxiang@ncsu.edu

Mike Mantor
Advanced Micro Devices
Orlando, FL
Michael.Mantor@amd.com

Norm Rubin
Advanced Micro Devices
Boxborough, MA
Norman.Rubin@amd.com

Huiyang Zhou
North Carolina State University
Raleigh, NC
hzhou@ncsu.edu

ABSTRACT
On-chip shared memory (a.k.a. local data share) is a critical resource to many GPGPU applications. In current GPUs, the shared memory is allocated when a thread block (also called a workgroup) is dispatched to a streaming multiprocessor (SM) and is released when the thread block is completed. As a result, the limited capacity of shared memory becomes a bottleneck for a GPU to host a high number of thread blocks, limiting the otherwise available thread-level parallelism (TLP). In this paper, we propose software and/or hardware approaches to multiplex the shared memory among multiple thread blocks.

Our proposed approaches are based on our observation that the current shared memory management reserves shared memory too conservatively, for the entire lifetime of a thread block. If the shared memory is allocated only when it is actually used and freed immediately after, more thread blocks can be hosted in an SM without increasing the shared memory capacity. We propose three software approaches to enable shared memory multiplexing and implement them using a source-to-source compiler. The experimental results show that our proposed software approaches effectively improve the throughput of many GPGPU applications on both NVIDIA GTX285 and GTX480 GPUs (an average of 1.44X on GTX285, 1.70X on GTX480 with 16kB shared memory, and 1.26X on GTX480 with 48kB shared memory). We also propose hardware support for shared memory multiplexing, which incurs minor hardware changes to existing hardware and enables significant performance improvements (an average of 1.53X) to be achieved with very little change in GPGPU code.

Categories and Subject Descriptors
B.3.2 [**Memory Structures**]: Design Styles – *shared memory*

General Terms
Performance, Design, Experimentation.

Keywords
GPGPU, Shared memory, Dynamic management.

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.

1. INTRODUCTION
Modern many-core graphics processor units (GPUs) rely on thread-level parallelism (TLP) to deliver high computational throughput. To mitigate the impact of long latency memory accesses, besides TLP, software managed on-chip local memory is included in state-of-art GPUs. Such local memory, called shared memory in NVIDIA GPUs and local data share in AMD GPUs, has limited capacity. As a shared resource among threads, shared memory is one of the key factors to determine how many threads can run concurrently on a GPU.

State-of-art GPUs manage shared memory in a relatively simple manner. When a group of threads (called a thread block or workgroup) is to be dispatched, the shared memory is allocated based on the aggregate shared memory usage of all the threads in the thread block (TB). When a TB finishes execution, the allocated shared memory is released. When there is not sufficient shared memory for a thread block, the TB dispatcher is halted.

There are two major limitations to the aforementioned shared memory management. First, the allocated shared memory is reserved throughout the lifetime of a TB, even if it is only utilized during a small portion of the execution time. Second, when the shared memory size (e.g., 16kB) is not a multiple of the shared memory usage (e.g., 9kB) of a TB, a fraction of shared memory (e.g., 7kB) is always wasted. These two limitations reduce the number of TBs that can concurrently run on a GPU, which may impact the performance significantly as there may not be sufficient threads to hide long latencies of operations such as memory accesses.

In this paper, we first characterize the usage of shared memory in GPGPU (general purpose computation on GPUs) applications and make an important observation that many GPGPU applications only utilize shared memory for a small amount of time compared to the lifetime of a TB. Then, we propose novel ways to multiplex shared memory so as to enable a higher number of TBs to be executed concurrently. These schemes include three software approaches, namely VTB, VTB_pipe, CO-VTB, and one hardware solution. Our software approaches work on existing GPUs and they essentially combine two original TBs into a new TB and add if-statements to control time multiplexing of the allocated shared memory between the two original TBs. Our hardware solution incurs minor changes to existing hardware and supports dynamic shared memory allocation and de-allocation so as to enable shared memory multiplexing with very little change in GPGPU code.

Our experimental results on NVIDIA GTX285 and GTX480 GPUs show remarkable performance gains from our proposed software approaches, up to 1.95X and 1.44X on average on GTX285 and up to 2.19X and 1.70X on average on GTX480. We also evaluate our hardware proposal using the GPGPUsim simulator [1], which shows up to 2.34X and an average of 1.53X performance enhancement.

Our work makes the following contributions. (1) We characterize shared memory usage among GPGPU applications and highlight that many GPGPU applications utilize shared memory only for a limited portion (an average of 25.6%) of the execution time of a TB; (2) we propose three software approaches to time multiplex shared memory among TBs; (3) we propose a simple hardware approach to support dynamic shared memory management; and (4) we show that our proposed software- and hardware-based shared memory multiplexing approaches are highly effective and significantly improve the performance.

The remainder of the paper is organized as follows. In Section 2, we present a brief background on GPGPU architecture and highlight the importance of shared memory. We characterize the shared memory usage of GPGPU applications in Section 3. In Section 4, we present our three software approaches to multiplex shared memory. Section 5 discusses our hardware solution. The experimental methodology is addressed in the Section 6 and the results are presented in Section 7. Related work is discussed in Section 8. Section 9 concludes our paper.

2. BACKGROUND

State-of-art GPUs use many-core architecture to deliver high computational throughput. One GPU consists of multiple streaming multiprocessors (SMs) in NVIDIA GPU architecture or computer units (CUs) in AMD GPU architecture. Each SM/CU in turn includes multiple streaming processors (SPs) or thread processors (TPs). Threads running on GPUs follow the single program multiple data (SPMD) model and are organized in a hierarchy. A GPU kernel is launched to a GPU with a grid of thread blocks (TBs) using the NVIDIA CUDA terminology [10], which are called workgroups in OpenCL [12]. Threads in a TB form multiple warps, with each running in the Single Instruction Multiple Data (SIMD) mode. One or more TBs run concurrently on one SM, depending on the resource requirement of a TB.

On-chip shared memory is a critical resource for GPGPU applications. The shared memory provides a mechanism for threads in the same TB to communicate with each other. It also serves as a software managed cache so as to reduce the impact of long latency memory accesses. Since each SM has limited amount of shared memory, for many GPGPU applications, the usage of shared memory of a TB determines how many TBs can run concurrently, i.e., the degree of thread level parallelism (TLP). Besides shared memory, the register usage of each thread is another critical factor to determine the number of threads that can run concurrently. In state-of-art GPUs, both shared memory and register files (RFs) are managed similarly. When a TB is to be dispatched to an SM, the TB dispatcher allocates the shared memory and registers based on the aggregate usage of all the threads in the TB. The allocated shared memory and registers are released when the TB finishes execution. When there is not sufficient resource available in either shared memory or RF in an SM, the resource is not allocated and no TB will be dispatched to the SM.

Between shared memory and RFs, current GPUs have higher capacity in RFs. For example, on NVIDIA GTX285 GPUs, each SM has 16kB shared memory and a 64kB RF. On NVIDIA GTX480 GPUs (i.e., the Fermi architecture), each SM has a 128kB RF and a 64kB hybrid storage that can be configured as a 16kB L1 cache+48kB shared memory or a 48kB L1cache+16kB shared memory. The latest NVIDIA GPU, GTX680 (i.e., the Kepler architecture), has the same size of shared memory per SM as GTX480 and a 256kB RF. With a high number of SPs and a larger RF in each SM, the Kepler architecture is designed to host more concurrent thread blocks/threads in each SM than the Fermi architecture, thereby increasing the pressure on shared memory. On AMD HD5870 GPUs, each CU contains 32kB shared memory (called local data share) and a 256kB RF. As a result, for many GPGPU applications, shared memory presents a more critical resource to limit the number of TBs/threads to run concurrently on an SM.

3. CHARACTERIZATION OF SHARED MEMORY USAGE

To understand how GPGPU applications utilize shared memory, we select and study ten benchmarks, which have shared memory variables in the source code, as shown in Table 1. Among the benchmarks, MC, SP, MM, CV, RD and TP are from NVIDIA SDK [11]. FFT and HG are from AMD SDK [2]. STO [13] is from the GPGPUSim infrastructure [1]. We implemented the GPU kernel for MV, which has similar performance to CUBLAS 4.0 [8]. For each benchmark, the shared memory usage of a TB as well as the number of threads in a TB is reported in Table 1. Also included is the number of TBs that can run concurrently in an SM with 16kB shared memory.

Table 1. Benchmarks used in experiments

Benchmarks	Shared memory size per TB (Bytes)	Threads per TB	Threads (TBs) per SM	Actual shared memory usage per SM (Bytes)
Matrix vector multiplication (MV)	4268	32	96 (3)	4268x3 = 12804
Fast Fourier Transform (FFT)	8736	64	64 (1)	8736x1 = 8736
MarchingCubes (MC)	9324	32	32 (1)	9324x1 = 9324
StoreGPU (STO)	16304	128	128 (1)	16304x1 = 16304
ScalarProd (SP)	4144	64	192 (3)	4144x3 = 12432
Histogram (HG)	8224	64	64 (1)	8224x1 = 8224
Convolution(CV)	8300	128	128 (1)	8300x1 = 8300
Matrix Multiplication (MM)	2084	256	1024 (4)	2084x4 = 8336
Transpose (TP)	4260	128	384 (3)	4260x3 = 12870
Reduction (RD)	540	128	1024 (8)	540x8 = 4320

From Table 1, we can classify the benchmarks into two categories. The first category, including MV, FFT, MC, STO, SP, HG and CV, has the characteristics that the number of threads, which can execute concurrently in a SM, is severely limited by the shared memory capacity. The second category, including MM, TP, and RD, has the characteristics that either the shared memory usage of each TB is small or each TB has a large number of threads. For the benchmarks in the second category, the shared memory is not a bottleneck for TLP. Therefore, the target of our proposed approaches is the workloads in the first category. Another interesting observation from Table 1 is that for all workloads in

the first category, except STO, the shared memory can be severely underutilized, as shown in the last column of Table 1, although the limited shared memory size is the cause for limited thread-level parallelism (TLP). The reason is that when a TB is to be dispatched, all its required shared memory needs to be available. Therefore, if the remaining shared memory is not enough for a TB, it is always wasted.

Next, for all the benchmarks in Table 1, we use simulation to determine how long the shared memory is utilized in a TB (see Section 6 for the detailed experimental methodology). Since the compiler may schedule the shared memory accesses to interleave with other type instructions to improve instruction-level parallelism (ILP), if we simply consider the lifetime of shared memory usage as between the first instruction writing to the shared memory and the last instruction reading from the shared memory, we may find that the shared memory is used for almost the entire lifetime of kernel execution. To isolate the usage of shared memory from other parts of the kernel code, we insert '__syncthreads()' instructions before the first *write/define* to and after the last *read/use* from the shared memory. We denote a code region surrounded by our inserted '__syncthreads()' as a shared memory access region. A redefine of the shared memory variables will start a new shared memory access region. Here, any define or use of shared memory variables is based on all the threads in a TB. Then, we use the accumulated execution time of all shared memory access regions as the duration for shared memory usage. In Figure 1, we show the ratio of the duration of shared memory usage over the overall execution time of a TB. For each benchmark, this ratio is an average across all its TBs.

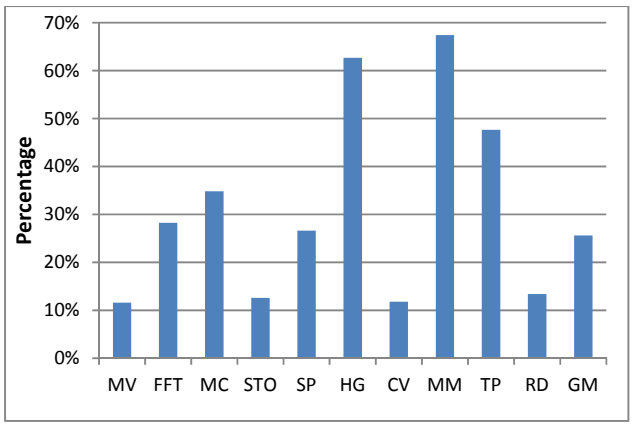

Figure 1. The portion of the execution time, during which the allocated shared memory is actually utilized, of a thread block for GPGPU applications.

As shown in Figure 1, using the geometric mean (GM) for all the benchmarks, the shared memory is only used in 25.6% of the execution time, which means that TBs do not need to use the allocated shared memory during the 74.4% of their execution time. Among these benchmarks, MM and TP have high TLP and they use the shared memory often during the lifetime of a TB. RD has high TLP but its performance bottleneck is global memory accesses; therefore the shared memory is idle most of the time. HG spends most of the execution time on accessing data in the shared memory, thereby showing the high ratio in Figure 1. For the remaining benchmarks, the TLP is limited by the shared memory capacity although their allocated shared memory is only used for a very limited amount of time compared to the lifetime of a TB.

4. SHARED MEMORY MULTIPLEXING: SOFTWARE APPROACHES

As discussed in Section 3, many GPGPU applications suffer from insufficient TLP due to the limited shared memory capacity. In this section, we propose three software approaches to time multiplex shared memory to boost TLP. The key idea of these three approaches is the same: we combine multiple original TBs into a larger one and introduce control flow to manage how the shared memory is accessed among the original TBs. The difference among the three approaches lies in how to overlap shared memory accesses with other parts of the code and whether the combined TB will use more shared memory than an original TB. To illustrate the proposed approaches, we use the FFT as a running example. Figure 2 shows the pseudo code of the kernel function, which implements a 1K-point FFT through a sequence of 4-point FFT (FFT4 functions) and data interchange through shared memory (loadFromSM and saveToSM functions). In loadFromSM, threads load data from shared memory. In saveToSM threads save data to the shared memory. '__syncthreads()' is used to ensure the order of the shared memory accesses. We also include the sequence number as a parameter of FFT4, loadFromSM and saveToSM functions to show the different parts of the code. With this implementation, each TB has 64 threads and uses 8736-Byte shared memory (8192 Bytes for data, additional bytes for padding to avoid bank conflicts and a few bytes reserved by CUDA). This shared memory usage is obtained from the NVCC compiler.

```
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();
```

Figure 2. The pseudo code of a 1k-point FFT implementation, which uses 8736-Byte shared memory per thread block and there are 64 threads per thread block.

4.1 Virtual Thread Block (VTB)

In this approach, we first isolate the part(s) of a kernel function that accesses shared memory variables. Second, we combine two original TBs into a new one. Here, we refer to an original TB as a virtual TB. In other words, after TB combination, one TB contains two virtual TBs. Third, we introduce the control flow "if(v_tb_id==0)" and "if(v_tb_id ==1)" to manage which virtual TB will access the shared memory at a time. The amount of the required shared memory of the combined TB remains the same as either of the virtual TBs.

For the FFT code example, the code after we apply VTB is shown in Figure 3. The 'if-statements' on lines 4, 6, 8, and 10 are introduced to ensure that only one virtual TB is accessing the allocated shared memory at a time. The 'syncthreads()' function on line 7 implicitly marks the last use of the shared memory of

virtual TB 0 so that virtual TB 1 can use the shared memory immediately afterwards.

```
1. int v_tb_id = threadIdx.x/64;  //virtual thread block id
2. loadFromGlobal();
3. FFT4(0);
4. if (v_tb_id==0) saveToSM(0); //def. from threads in v_tb_0
5. __syncthreads();
6. if (v_tb_id==0) loadFromSM(0);//use. from threads in v_tb_0
7. __syncthreads();
8. if (v_tb_id==1) saveToSM(0);  //def. from threads in v_tb_1
9. __syncthreads();
10.if (v_tb_id==1) loadFromSM(0);//use. from threads in v_tb_1
11.FFT4(1);
12.     ….
13.FFT4(4);
14.writeToGlobal();
```

Figure 3. The pseudo code of a 1k-point FFT implementation using VTB. Each thread block uses 8736-Byte shared memory and there are 128 threads in each thread block.

Next, we illustrate the reason why our proposed VTB can improve the GPU throughput and also highlight its overhead. Assuming a GPU with 16kB shared memory in each SM, since each TB requires more than 8kB shared memory, two TB dispatched to the same SM have to execute back to back with the code in Figure 2. This execution process is shown in Figure 4a. For the purpose of clarity, in Figure 4 we only show the execution time corresponding to the global memory access, the first 4-point FFT and the data exchange via the shared memory. The remaining code in the kernel function simply repeats 4-point FFT and data exchange multiple times. With the code in Figure 3, the combined TB is equivalent to the two original TBs. Due to the increased TLP, the execution time of the function loadFromGlobal() and FFT4() of 128 threads is significantly less than the back-to-back execution of the same functions for 64 threads, as shown in Figure 4b. However, to control the accesses to shared memory between the two virtual TBs, additional synchronization functions are added to ensure correctness. Besides the latency to perform such '__syncthread()' functions, the barrier also limits the compiler's capability to schedule instructions across the barriers, which may result in reduced instruction-level parallelism (ILP) and additional register usage. The added control flow instruction "if(v_tb_id==0)" has minimal overhead as it does not generate any control divergence within a warp since all 64 threads in the same virtual TB will follow the same direction and each warp has 32 threads on NVIDIA GPUs. The global memory access functions 'loadFromGlobal' and 'writeToGlobal' benefit from VTB as the increased TLP translate to increased memory-level parallelism (MLP).

From Figure 4, we can also see that when a virtual TB accesses the shared memory, the other virtual TB is forced to be idle due to the control flow and the '__syncthread()' functions. Our proposed second and third software approaches address this limitation and we include the execution time information of these approaches in Figure 4c and 4d for comparison. We discuss these two approaches in detail in Sections 4.2 and 4.3.

Note that although Figure 3 and Figure 4 show the case of combining two original TBs into one, we can apply the same principle to combine more than two TBs. The optimal number of TBs to combine is dependent on how many concurrent threads can run on an SM. Typically, combining two TBs is sufficient to reap most performance benefits. Among all the benchmarks in our study, only MarchingCubes (MC) benefits from combining more than two TBs (we combined 4 TBs for MC using our proposed VTB approach).

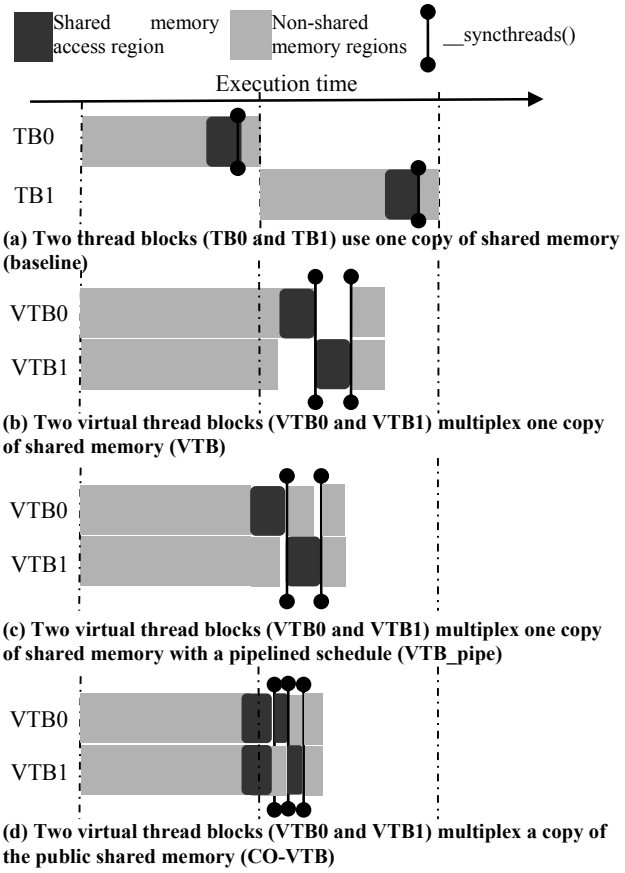

Figure 4. A comparison of execution time of the baseline to our proposed software approaches: (a) the baseline, (b) VTB, (c) VTB_pipe, and (d) CO-VTB.

4.2 Pipelined Virtual Thread Block (VTB_PIPE)

As discussed in Section 4.1, VTB combines two virtual TBs into a larger one and it ensures that only one virtual TB is accessing the shared memory by forcing the other virtual TB to be idle. To reduce such idle cycles, we propose to overlap computation with shared memory accesses. To do so, we make the first virtual TB to run faster than the second one using an 'if(v_tb_id==0)' statement. Then, when the first virtual block reaches the code section of shared memory access, the second virtual TB continues its computation instead of being forced idle. When the second virtual block reaches the code section of shared memory accesses, the first will continue to run ahead. This process is similar to letting the two virtual TBs to go through a pipeline. Therefore, we refer to this approach as pipelined VTB (VTB_pipe).

For the FFT example, the code after we apply our proposed VTB_pipe is shown in Figure 5. From Figure 5, we can see that initially the two virtual TBs will both execute the 'loadFromGlobal()' function. Then, the 'if(v_td_id==0)' statements on lines 3 and 4 as well as the '__syncthreads()' on line 5 enable virtual TB 0 to execute the 'FFT4()' and 'saveToSM'

functions, making it running ahead of virtual TB1. The code on line 6 and line 7 shows the overlapping between the function 'loadFromSM()' of virtual TB0 and the 'FFT4()' function of virtual TB1. Since virtual TB1 is lagging behind, when it reads from the shared memory via 'loadFromSM()' on line 11, the virtual TB0 proceeds to compute its next 4-point FFT, the 'FFT4()' on line 12. The execution process is shown in Figure 4b. Due to the overlapping between shared memory accesses and computation, we can reduce the idle cycles experienced by virtual TBs.

```
1.  int v_tb_id = threadIdx.x/64;
2.  loadFromGlobal();
3.  if (v_tb_id==0)  FFT4(0);
4.  if (v_tb_id==0) saveToSM(0);
5.  __syncthreads();
6.  if (v_tb_id==0) loadFromSM(0);
7.  else FFT4(0);
8.  __syncthreads();
9.   if (v_tb_id==1) saveToSM(0);
10. __syncthreads();
11. if (v_tb_id==1) loadFromSM(0);
12. else FFT4(1);
13. __syncthreads();
14. if (v_tb_id==0) saveToSM(1);
15. __syncthreads();
16. if (v_tb_id==0) loadFromSM(1);
17. else FFT4(1);
18.  ….
19. FFT4(4);
20. writeToGlobal();
```

Figure 5. The pseudo code of a 1k-point FFT implementation using VTB_pipe. Each thread block uses 8736-Byte shared memory and there are 128 threads in each thread block.

The complexity of VTB_pipe, however, is that we may need to partition the non-shared memory access code section to create small computational/global memory access tasks so that they can overlap with shared memory accesses. The ideal case is that the small computational tasks have similar execution latency to the shared memory accesses and can completely utilize the otherwise idle cycles. In the FFT example, the FFT4 is a convenient choice and does not require such partition. For other benchmarks such as Histogram (HG), loop peeling is used to create such a computational/global memory access task to overlap with the shared memory accesses. Similar to the VTB approach, we can choose to combine more than two original TBs. However, synchronization among more than two TBs becomes difficult to manage. Therefore, we choose not to combine more than two TBs for our VTB_pipe approach.

4.3 Collaborative Virtual Thread Block (CO_VTB)

In both VTB and VTB_pipe, a few original TBs are combined to time multiplex the allocated shared memory, thereby significantly improving TLP. For some applications, the TLP improvement is sufficient to hide instruction execution latencies. For others, there exist additional opportunities. As discussed in Section 3, if the shared memory size is not a multiple of the shared memory usage of a TB, part of shared memory is always wasted. Neither VTB nor VTB_pipe addresses this issue as they do not alter the shared memory usage of a TB. To effectively utilize such otherwise wasted shared memory, we propose to let two TBs to collaboratively utilize the shared memory and refer to this approach as collaborative virtual thread blocks (CO-VTB). In CO-VTB, we partition the shared memory usage of a TB into two parts, private and public, and apply the VTB (or VTB_pipe) approach only on the public part. For example, if the original shared memory usage of a TB is 9kB, an SM with 16kB shared memory can only host 1 TB. If we partition the shared memory usage of a TB into a 7kB private part and a 2kB public part, when we combine two TBs, each uses 7kB private shared memory each (a total of 14kB) and both time multiplex the 2kB public shared memory, thereby utilizing the 16kB shared memory effectively. Figure 4d illustrates the execution of our proposed CO-VTB approach.

Next, we use the benchmark, MatchingCube, to show the code changes for CO-VTB. For the benchmark FFT, CO-VTB involves too much code change, which incurs high performance overhead. The simplified pseudo code of the baseline MatchingCube kernel is shown in Figure 6a. From the code, we can see that each TB has 32 threads and uses 9216-Byte (=24*32*3*4) shared memory. The code after we apply CO-VTB is shown in Figure 6b. Now, one TB has 2 virtual TBs and there are 64 threads in a TB. The shared memory array is partitioned into two parts: the private arrays 'vertlist_v0' and 'vertlist_v1', which are combined into a single array 'vertlist', and the public array 'vertlist2', which is multiplexed by the two virtual TBs. Either private array has the size of 6144(=16*32*3*4) Bytes and the public array size is 3072(=8*32*3*4) Bytes. So, the overall shared memory usage of a TB becomes 15360 (= 2*6144+3072) Bytes. The register variable 'reg' is introduced to temporarily hold the data to be written to the public part of the shared memory. Additional code is inserted to check the array index (the variable 'edge') to determine whether the data resides in the private or public part of shared memory and then either the array 'vertlist' (private) or the array 'vertlist2' (public) is used accordingly.

As shown from the code example in Figure 6, there is overhead involved in the CO-VTB approach, including additional register variables and additional code. For kernel functions like FFT, the complex array access patterns introduce too much overhead when a shared memory array is partitioned to private and public parts. Therefore, CO-VTB is utilized selectively, only for arrays with relatively simple access patterns. Due to this complexity, we also choose not to combine more than two TBs.

5. SHARED MEMORY MULTIPLEXING: A HARDWARE SOLUTION

As discussed in Section 4, our proposed software approaches improve TLP by merging original TBs and explicitly managing the shared memory accesses among them. The advantage of the software approaches is that they work well with current GPUs. The disadvantage, however, is the overhead introduced to manage the shared memory. In this section, we propose a hardware solution to managing shared memory.

In GPUs, the TB dispatcher dispatches TBs onto SMs. For each SM, it maintains a shared memory management (SMM) table, as shown in Figure 7. The SMM table has multiple entries and each entry keeps three fields, the TB id, the size, and the starting address. When a TB is to be dispatched to an SM, the TB dispatcher goes through the SMM table of the SM to determine whether there is enough free shared memory. If so, the dispatcher allocates the required shared memory by filling an entry in the SMM table with the TB id and setting its size field to the required shared memory size of the TB. The starting address field is

determined and then passed to the TB so that every shared memory access in the TB will use this starting address as the base address. When a TB finishes execution, the dispatcher releases the allocated shared memory by invalidate the corresponding SMM table entry. Since the shared memory is allocated through the lifetime of a TB, we refer to such shared memory management as 'static' allocation and de-allocation.

```
#define NTHREADS 32
__global__ void generateTriangles() {
  //each TB uses 4*3*24*32 = 9216 byte share memory.
   __shared__ float3 vertlist[24][NTHREADS];
   // store the result of function f1 into shared memory
   for (i=0; i<24; i++) vertlist [i][threadIdx.x]= f1(i);
   for(i=0; i<numVerts; i++) {
      uint edge = tex1Dfetch(triTex, cubeindex*16 + i);
      // use function f2 perform computation on shared memory
      pos[index] = f2(vertlist[edge][ threadIdx.x]);
  }
}
```

(a) The pseudo code of MarchingCubes, for which each thread block uses 9216-Byte shared memory and has 32 threads.

```
#define NTHREADS 32
__global__ void generateTriangles() {
// each virtual TB uses 6144-byte private shared memory.
//__shared__ float3 vertlist_v0[16][ NTHREADS];
//__shared__ float3 vertlist_v1[16][ NTHREADS];
//Both private parts are combined into a single array vertlist
    __shared__ float3 vertlist[2][16][NTHREADS];
// accessing the private part of shared memory
   for (i=0; i<16; i++) vertlist [v_tb_id][i][v_t_id]=f1(i);
   float3 reg[8];
   for (i=16; i<24; i++) reg[i-16] = f1(i);
   __syncthreads();
// two virtual TBs multiplex 3072-byte public shared memory.
   __shared__ float3 vertlist2[8][NTHREADS];
   if (v_tb_id==0) {
     for(i=0; i<8; i++) vertlist2[i][v_t_id]= reg[i];
     for(i=0; i<numVerts; i++) {
       uint edge = tex1Dfetch(triTex, cubeindex*16 + i);
// check whether the data is in private or public shared memory
      if( edge<16) pos[index] =
                  f2(vertlist[v_tb_id][edge][v_t_id])
      else pos[index] = f2(vertlist2[edge-16][v_t_id]);  } }
   __syncthreads();
   if (v_tb_id==1) { … }
}
```

(b) The pseudo code of MarchingCubes using CO-VTB. Each thread block uses 2*6144+3072=15360 Byte shared memory and there are 64 threads in each thread block.

Figure 6. The simplified pseudo code of MarchingCubes, (a) the baseline kernel code; (b) the code after we apply CO-VTB.

Valid	TB id	Size	Starting Address
1	6	4kB	0
1	10	4kB	4096
	…	…	

Figure 7. A shared memory management (SMM) table.

To enable dynamic shared memory management, we propose to extend the TB dispatcher so that the shared memory management is exposed to and can be controlled by software. Since shared memory allocation and de-allocation affect all the threads in a TB, we propose to associate shared memory management with the existing '__syncthreads()' function and the new syntax of the function becomes '*__syncthreads(int opt, unsigned &base_addr, unsigned size)*'. It still serves as a barrier to synchronize all the threads in a TB. When the parameter 'opt' is 1, it invokes the TB dispatcher to allocate the shared memory for 'size' bytes. The TB dispatcher uses the same allocation process by going through the corresponding SMM table. If there is enough free shared memory, an entry in the SMM table is updated and its 'starting address' field is passed to the 'base_addr' variable to be used by subsequent shared memory accesses. If there is no sufficient shared memory to be allocated, the TB will be stalled until another TB frees its allocated shared memory. If the parameter 'opt' is '-1', the TB dispatcher performs shared memory de-allocation using the 'base_addr' parameter. It searches the SMM table entries to find the matching 'starting address' with the 'base_addr' and invalidates the table entry. If the parameter 'opt' is '0', the other two parameters ('base_addr' and 'size') are ignored and '__syncthreads' operates as a regular barrier. To simplify the design, we choose not to allow nested shared memory allocation so as to avoid any potential deadlock issue. We also require that for a kernel function, the size of dynamic allocation and de-allocation to be the same so as to avoid fragmentation.

The code change to utilize our proposed dynamic shared memory management is very little. It only needs a shared memory allocation at the beginning and de-allocation at the end of each memory access code region. For the FFT kernel, the code after such changes is shown in in Figure 8.

```
loadFromGlobal();
FFT4(0);
__syncthreads(1, &base_addr , 8376); //allocation
saveToSM(0);       //define by multiple threads in a TB
__syncthreads(0, 0, 0); //synchronization
loadFromSM(0);   //use by multiple threads in a TB
__syncthreads(-1, &base_addr , 8376); //de-allocation

FFT4(1);
….
writeToGlobal();
```

Figure 8. The pseudo code for 1k-point FFT kernel using the hardware supported dynamic shared memory management. Each TB allocates and de-allocates 8376-byte shared memory and contains 64 threads.

The dynamic shared memory allocation and de-allocation in the FFT kernel shown in Figure 8 enables an SM to exploit higher degrees of TLP. Figure 9 illustrates this effect with two TBs running on an SM. Although the shared memory (16kB) on the SM is not large enough for the aggregate requirement from the two TBs (2x8376=16752B), our proposed dynamic allocation enables them to run concurrently and ensures that the two allocation calls will be served one after the other.

From Figure 9, it can be seen that the key performance advantages of our dynamic shared memory management include: (1) higher degrees of TLP to hide instruction latencies, and (2) reduced overhead of '__syncthreads()' as the barrier is limited to a TB and doesn't affect other TBs. In comparison, in our software

approaches, such a barrier will affect both virtual TBs. Furthermore, dynamic allocation and de-allocation naturally enables overlap between shared memory accesses of one TB and non-shared memory code in another as long as they do not reach allocation at the same time. This is the reason why we do not need the code changes of our VTB or VTB_pipe approaches.

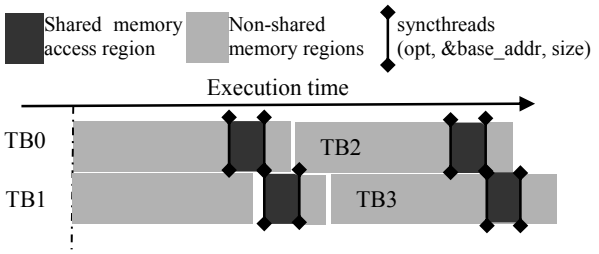

Figure 9. Two TBs running concurrently on an SM using dynamic shared memory management.

With the hardware supported shared memory management, we do not need to combine TBs. A TB can be dispatched to an SM as long as other resource requirements such as registers are satisfied. As a result, there might be too many TBs dispatched to an SM. We propose to use a counter to track how many TBs are running on an SM and stall TB dispatching when this counter reaches a threshold. If we denote the maximum number of TBs that can be dispatched to an SM using the static shared memory management as *K*, the threshold setting of *K+2* or *K+3* achieves good performance (See Section 7.2). In other words, allowing an SM to run 2 or 3 more TBs concurrently usually improves TLP sufficiently.

With our proposed hardware solution, the TB dispatcher can support both static and dynamic shared memory management. We propose to let either the run-time or compiler to determine which mechanism to be used. If the static management is selected for the purpose of quality of service, the dynamic management instructions are ignored. Static and dynamic shared memory management can also be used together to support collaborative TBs, similar to the idea exploited in CO-VTB. We refer to this hardware supported collaborative TB approach as CO-HW. Like CO-VTB, we need to change the kernel code to partition the shared memory usage into the private part and the public part. Unlike CO-VTB, we do not need to use VTB or CO-VTB on the public part. Instead, we insert dynamic allocation and de-allocation instruction to multiplex the public part. When a kernel is launched to a GPU, the compiler or the run-time provides the sizes of both the private part and the public part to the TB dispatcher. Static shared memory management is used for the private part and dynamic shared memory management is used to multiplex the public part. For example, a TB originally uses 6kB shared memory and an SM with 16kB shared memory can host two such TBs using static shared memory management. After partition, a TB uses 4kB private and 2kB public shared memory. With 16kB shared memory, three TBs can run concurrently using a total of 12kB (=4kBx3) private shared memory. The remaining 4kB is used as public shared memory among the three TBs. The same SMM table is used to manage shared memory as shown in Figure 10, where the first three entries are allocated when the three TBs are dispatched and the last two are allocated/de-allocated with the '__syncthreads(opt, &base_addr, size)' instructions.

Valid	TB id	Size	Starting Address
1	0	4kB	0
1	1	4kB	4096
1	2	4kB	8192
1	0	2kB	12288
1	1	2kB	14436

Figure 10. The SMM table manages shared memory usage for collaborative TBs. The first three entries are for the private shared memory of the 3 TBs. The last two entries are for the public shared memory.

Due to static management, the private part is allocated when a TB is dispatched and is de-allocated when it completes execution. The public part is managed based on the dynamic allocation and de-allocation instructions. The only constraint, which the TB dispatcher enforces, is that when a TB is dispatched, the total amount of the private parts of currently running TBs in an SM cannot exceed (the size of overall shared memory – the size of the public part of a TB). The purpose is to ensure that there is at least one set of public shared memory available to be used among the TBs.

Our proposed hardware solution simply exposes the existing shared memory management in the TB dispatcher to the software and enables it to be controlled by the extended '__syncthreads()' instructions. The code change is to insert '__syncthreads(1, &base_addr, size)' at the beginning of shared memory access regions and '__syncthreads(-1, &base_addr, size)' at the end. For CO-HW, this region is where the public shared memory part is accessed. Therefore, we argue that this solution has low overhead in both hardware and software changes. The effectiveness is evaluated in Section 7.2.

6. EXPERIMENTAL METHODOLOGY

To evaluate our proposed software approaches, we use both NVIDIA GTX 480 and NVIDIA GTX285 GPUs with CUDA SDK 4.0. Because the shared memory size is configurable on GTX 480 GPUs, we present two sets of results: one with 48kB shared memory and the other with 16kB shared memory. As discussed in Section 3, the focus of our proposed approaches is the category of applications, which have low TLP due to the limited capacity of shared memory in an SM. Therefore, among all the workloads in Table 1, our experiments do not include MM, TP, and RD as they have high TLP already and are not affected by our approaches. We implemented our proposed VTB and VTB_pipe using a source-to-source compiler, Cetus [14]. For VTB, the compiler searches defines and uses of shared memory variables among all the threads in a TB. The code between the first define and the last use is treated as a shared memory access region. Re-defines to shared memory variables are used to help to determine the last uses in a region. Then, the compiler generates the code for virtual TB ids and the control flow to determine which virtual TB accesses the shared memory. For VTB_pipe, we add annotations manually to denote the section of code to overlap with shared memory access regions and the compiler generates the final code. For CO-VTB, we generate the code manually due to the complexity of partitioning shared memory variables into private and public parts. Also, as discussed in Section 5.3, not all workloads are suitable to CO-VTB due to the associated overheads. Among the workloads, we applied CO-VTB to MC, STO, and HG.

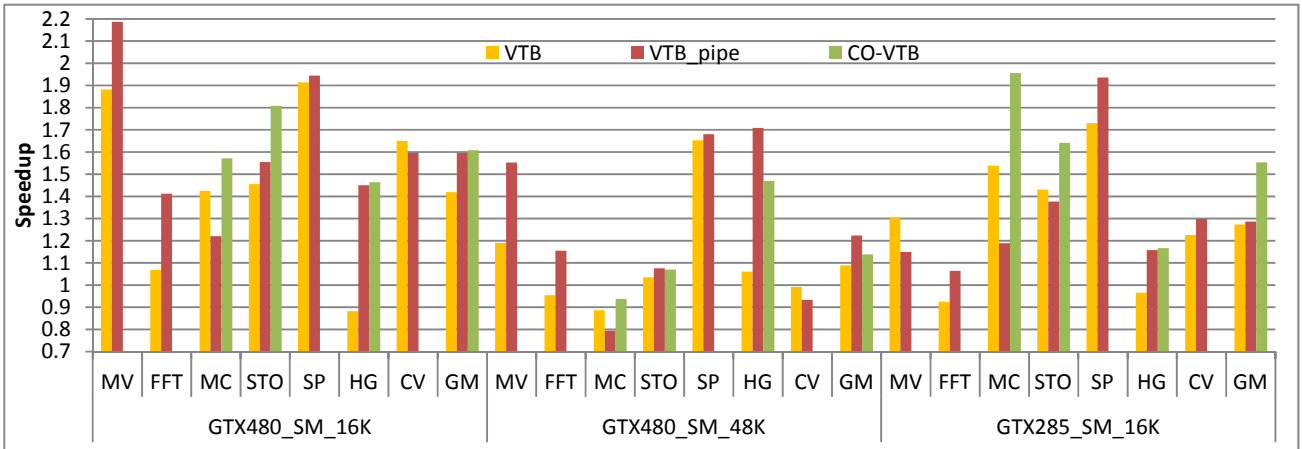

Figure 11. Speedups of the proposed software approaches over the baseline for different GPUs.

To model our proposed hardware support, we extend the GPGPUsim V3.0 simulator [1] to support our proposed dynamic shared memory management instructions (i.e., the extended '__syncthreads()' instruction). The simulator models an NVIDIA GTX285 GPU, which has 16kB shared memory and a 64kB register file on each SM. The off-chip memory frequency is set to 1100MHz. We manually inserted the dynamic allocation and de-allocation instructions in the workloads. For the CO-HW approach, we used very similar code to CO-VTB except the part using virtual TB ids as we do not need to combine two TBs. We also inserted the dynamic allocation and de-allocation instructions surrounding the code region that accesses the public shared memory.

7. EXPERIMENTAL RESULTS

7.1 Evaluation of Software-Based Shared Memory Multiplexing

In our first experiment, we evaluate the performance improvements from our proposed software approaches, VTB, VTB_pipe and CO-VTB. In Figure 11, we report the speedups of these three approaches over the baseline implementation. For each benchmark, we examine three GPU configurations, GTX 480 with 16kB shared memory (labeled "GTX480_SM_16K"), GTX 480 with 48kB shared memory (labeled "GTX480_SM_48K") and GTX 285 with 16kB shared memory (labeled "GTX285_SM_16K"). From the figure, we can make some interesting observations. First, among the three software approaches, CO-VTB achieves highest performance on average, using geometric mean (GM). However, as discussed in Section 6, not all workloads are suitable for this approach. VTB_pipe achieves higher performance than VTB on average as it can overlap shared memory accesses with other parts of code. The benchmark, MC, is an exception since the shared memory accesses dominate its execution and the overlapping effect from VTB_pipe fails to offset the overhead. Since different workloads may favor different shared memory multiplexing approaches, in order to achieve the best performance, we can generate three versions of optimized code using VTB, VTB_pipe and CO-VTB and select the best performing one. Second, among three GPU configurations, highest performance gains are achieved on GTX 480 with 16kB shared memory and the average speedups are 1.42X, 1.60X, 1.61X for VTB, VTB_pipe and CO-VTB, respectively. If we select the best version for each benchmark, the average performance gain reaches 1.70X. The reason is that compared to GTX285, GTX480 has a higher number of SPs. One instruction from a warp will keep the SP busy for 4 cycles on GTX285 compared 2 cycles on GTX480. Therefore, compared to GTX285, TLP is more critical for GTX480. Third, the increased shared memory capacity on GTX480 with 48kB shared memory enables an SM to host more TBs. The improved TLP in turn reduces the benefit of our proposed software schemes. For example, for the benchmark CV, each TB has 128 threads and uses 8300 Bytes of shared memory. As a result, 16kB shared memory can only host 1 TB while 48kB shared memory can host 5TBs (or 640 threads). Nevertheless, our proposed approaches remain effective and the achieved speedups are 1.09X, 1.22X, 1.14X on average for VTB, VTB_pipe, and CO-VTB, respectively. Selecting the best version for each benchmark provides a 1.26X speedup on average.

As two of our benchmarks, FFT and MV, are implemented in NVIDIA libraries, we compare our implementation of FFT to CUFFT4.0 and MV to CUBLAS4.0 on GTX480. For MV, we keep the width of the input matrix as 1024 and vary the height from 8K to 128K. The reason is that the height of the input matrix determines the number of threads for MV. The throughput comparison to CUBLAS4.0 is shown in Figure 12.

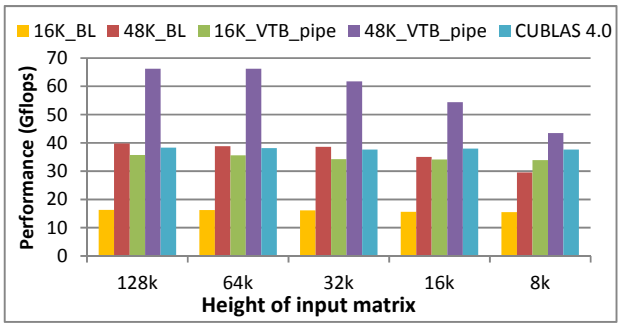

Figure 12. Performance comparison of MV among the baseline (xK_BL), VTB_pipe (xK_VTB_pipe) and CUBLAS 4.0 on GTX 480. 'xK' denotes the size of shared memory.

In Figure 12, the results with label '16K_BL' and '48K_BL' are our baseline implementation running on GTX480 with 16kB shared memory and 48kB shared memory, respectively. Our VTB_pipe results are labeled as '16K_VTB_pipe' and '48K_VTB_pipe' for the two shared memory configurations of GTX480. From the results, we can see that our baseline implementation running on GTX 480 with 48kB shared memory

has similar performance to CUBLAS. With our VTB_pipe approach on GTX480 with 16kB shared memory, we can achieve similar performance to CUBLAS. On the GTX480 configuration with 48kB shared memory, our VTB_pipe approach outperforms CUBLAS by up to 74% and 52% on average.

For 1K-point FFT, we use batch execution [7] to evaluate the throughput and vary the batch size from 128 to 2048. The throughput results are reported in Figure 13.

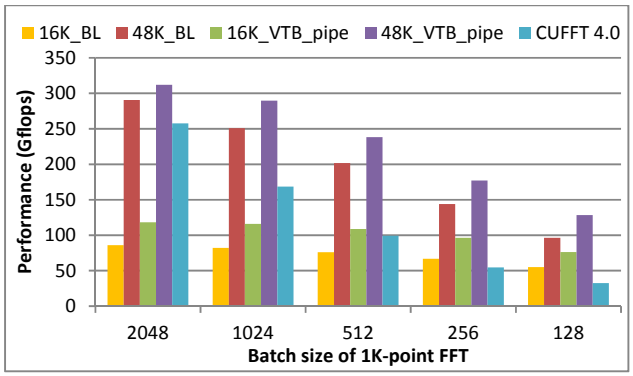

Figure 13. Performance comparison of FFT among the baseline (xK_BL), VTB_pipe (xK_VTB_pipe) and CUFFT 4.0 on GTX 480. 'xK' denotes the size of shared memory.

From the figure, we can see that our baseline implementation running on GTX480 with 16kB shared memory outperforms CUFFT [9] for small batch sizes and not as good as CUFFT for large batch sizes. With the 48kB shared configuration, our baseline implementation consistently outperforms CUFFT. The average throughput of '48K_BL' is 168.3 GFLOPS compared to the average of 72.4 GFLOPS throughput of CUFFT. Our VTB_pipe further improves the throughput by up to 33% and achieves the average throughput of 205.9 GFLOS (a 2.84X speedup over CUFFT).

7.2 Evaluation of Hardware-Supported Shared Memory Multiplexing

To evaluate the effectiveness of our proposed hardware solution, we first measure the performance of the baseline implementation (i.e., the static shared memory management) and the one with hardware support for dynamic shared memory management. Here, we use execution time (in the unit of cycles) rather than instruction per cycle (IPC) since we insert the dynamic allocation and de-allocation instructions into the code. The speedups of our hardware-supported dynamic shared memory management (labeled 'HW') over the baseline are shown in Figure 14. We also report the performance results of CO-HW, in which the shared memory usage is partitioned into statically managed private and dynamically managed public parts, in the figure (labeled 'CO_HW').

From Figure 14, we can see that our hardware supported dynamic allocation and de-allocation can significantly improve the performance, up to 2.34X and 1.53X on average, over the baseline static shared memory allocation. Similar to the software-based CO-VTB approach, we manually modified the code of MC, STO and HG for CO-HW. From Figure 14, it can be seen that CO-HW achieves up to 1.88X and an average of 1.42X speedups over the baseline. As discussed in Section 5, our hardware support dynamic shared memory management ('HW') eliminates some overheads of VTB or VTB_pipe. As a result, between HW and CO-HW, CO-HW remains more effective for MC but not for STO and HG. In contrast, for their software counterparts, CO-VTB typically performs much better than VTB or VTB-pipe.

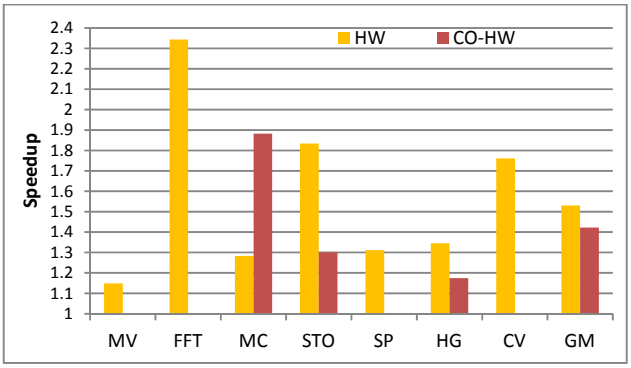

Figure 14. Speedups of hardware-supported dynamic shared memory management over baseline.

As discussed in Section 5, using dynamic shared memory management, many TBs can be dispatched to an SM if the shared memory is the only resource bottleneck. This may generate too much contention for multiplexing shared memory. A counter scheme is proposed in Section 5 to control the number of concurrent TBs that can be dispatched to an SM. If we use *K* to denote the maximum number of TBs that can be supported using the static shared memory management, we vary the upper bound of this counter from *K+1* to *K+3* and show the performance impact in Figure 15. In other words, we use dynamic shared memory management to allow an SM to host 1~3 more TBs. From the figure, we can see that on average, hosting 1 more TB ('K+1') does not provide sufficient TLP. Although hosting 2 or 3 more TBs in an SM shows similar performance, individual benchmarks show different trends. FFT and MC favor more TLP while HG and MV do not. Overall, our results suggest that either *K+2* or *K+3* is a fine choice as the maximum number of TBs to be allowed to run concurrently on an SM.

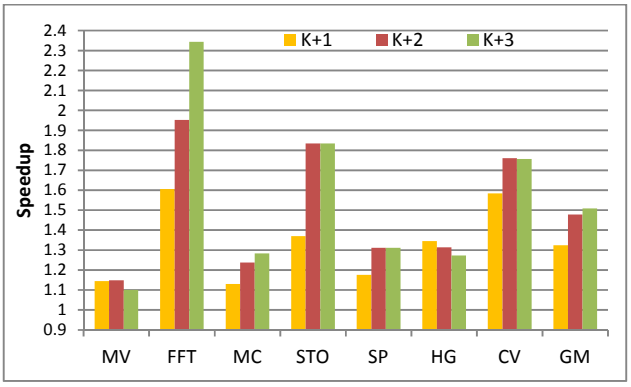

Figure 15. The impact of the maximum number of TBs to be allowed to run in an SM.

8. RELATED WORK

On-chip shared memory is a critical resource for GPGPU applications. Previous works mainly focus on utilizing shared memory to achieve coalesced memory accesses [3][4][5][15][16][17][18][19][20], to provide data exchange among threads [7], to use shared memory as software managed cache [18], etc. Although it is well known that heavy usage of shared memory may limit TLP [6][18], it is common that the

benefits of using shared memory overweigh the shortcomings of reduced TLP. As a result, many GPGPU workloads as shown in Section 3 have exhibited high shared memory usage. This is also a reason why the latest NVIDIA Fermi architecture (e.g., GTX 480 GPUs) provides larger shared memory and an L1 cache. However, as shown in Section 7, the high number of SPs in an SM in GTX480 (and even higher number of SPs in GTX680) makes TLP more important to hide instruction execution latencies. In contrast, our work improves TLP without sacrificing the usage of shared memory.

9. CONCLUSION

In this paper, we propose novel software and hardware approaches to multiplex shared memory. Our approaches are based on our observation that for the GPGPU applications with heavy use of shared memory, the duration of time, when the shared memory is utilized, is actually low. Our experimental results confirm that the shared memory is utilized for only 25.6% of the execution time of a TB. Therefore, there exist significant opportunities to time multiplex shared memory. Among our software approaches, VTB is simplest and it combines two TBs into a new one and adds control flow to ensure only one original TB accesses the shared memory at a time. VTB_pipe reduces the performance overhead of VTB by overlapping non-shared memory access regions (e.g., computation or global memory accesses) with shared memory accesses. CO-VTB partitions the shared memory data into a private part and a public part and only applies VTB/VTB_pipe upon the public part. Our proposed hardware support essentially exposes the existing shared memory management to software and enables software to control when to perform allocation and de-allocation. Our experimental results show that our proposed software schemes improve the performance significantly on current GPUs. We evaluate our hardware solution using the GPGPUsim simulator and the results show that it improves the performance remarkably with very little change in GPGPU code.

10. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments to improve our paper. This work is supported by an NSF project 1216569, an NSF CAREER award CCF-0968667 and a gift fund from AMD Inc.

11. REFERENCES

[1] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In IEEE International Symposium on Performance Analysis of Systems and Software, April 2009.

[2] AMD Accelerated Parallel Processing SDK V2.3, 2011

[3] B. Jang, D. Schaa, P. Mistry and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. In IEEE Transactions on Parallel and Distributed Systems, 2010.

[4] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In International Conference on Architectural Support for Programming Languages and Operating Systems, 2011.

[5] G. Ruetsch and P. Micikevicius, Optimize matrix transpose in CUDA. NVIDIA, 2009.

[6] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, A Performance Analysis Framework for Identifying Performance Benefits in GPGPU Applications. In Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2012.

[7] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In Proc. Supercomputing, 2008.

[8] NVIDIA CUDA Toolkit 4.0 CUBLAS Library, 2011

[9] NVIDIA CUDA Toolkit 4.0 CUFFT Library, 2011

[10] NVIDIA CUDA C Programming Guide 4.0, 2011.

[11] NVIDIA GPU Computing SDK 4.0, 2011.

[12] OpenCL, http://www.khronos.org/opencl/

[13] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In International Symposium on High Performance Distributed Computing, 2008.

[14] S. I. Lee, T. Johnson, and R. Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In Workshops on Languages and Compilers for Parallel Computing, 2003

[15] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Optimization space pruning for a multi-threaded GPU. In Proc. International Symposium on Code Generation and Optimization, 2008.

[16] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008.

[17] V. Volkov and J. W. Benchmarking GPUs to tune dense linear algebra. In Proc. Supercomputing, 2008.

[18] Y. Yang, P. Xiang, J. Kong and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In ACM SIGPLAN conference on Programming Language Design and Implementation, 2010.

[19] Y. Yang, P. Xiang, M. Mantor and H. Zhou. Fixing Performance Bugs: An Empirical Study of Open-Source GPGPU Programs. In International Conference on Parallel Processing, 2012.

[20] Y. Zhang, J. Cohen, and J. D. Owens. Fast Tridiagonal Solvers on the GPU. In Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2010.