Automatic Data Placement into GPU On-Chip Memory Resources

Chao Li[#] Yi Yang* Zhen Lin[#] Huiyang Zhou[#]

[#] Department of Electrical and Computer Engineering, North Carolina State University
*Department of Computer Systems Architecture, NEC Labs
[#]{cli17, zlin4, hzhou}@ncsu.edu; *yyang@nec-labs.com

Abstract

Although graphics processing units (GPUs) rely on thread-level parallelism to hide long off-chip memory access latency, judicious utilization of on-chip memory resources, including register files, shared memory, and data caches, is critical to application performance. However, explicitly managing GPU on-chip memory resources is a non-trivial task for application developers. More importantly, as on-chip memory resources vary among different GPU generations, performance portability has become a daunting challenge.

In this paper, we tackle this problem with compiler-driven automatic data placement. We focus on programs that have already been reasonably optimized either manually by programmers or automatically by compiler tools. Our proposed compiler algorithms refine these programs by revising data placement across different types of GPU on-chip resources to achieve both performance enhancement and performance portability. Among 12 benchmarks in our study, our proposed compiler algorithm improves the performance by 1.76x on average on Nvidia GTX480, and by 1.61x on average on GTX680.

1. Introduction

Throughput-oriented architecture, such as graphic processing units (GPUs), has been widely used to accelerate many general-purpose computation workloads. Although general purpose computation on GPUs (GPGPU) achieves high throughput mainly by employing a large bundle of threads to overlap computations with long-latency memory accesses, off-chip memory bandwidth and latency remain a performance as well as energy-efficiency bottleneck. Furthermore, the current trend of GPGPU evolution scales the computational throughput much faster than off-chip memory access bandwidth. For example, Nvidia GTX480 GPUs based on the FERMI architecture [17] have an arithmetic throughput of 1.35 TFLOPSs with the memory bandwidth of 178 GB/s. In comparison, GTX680 GPUs based on the KEPLER architecture [18] have an arithmetic throughput 3.09 TFLOPS (2.3X increase over GTX480) with the memory bandwidth of 192 GB/s (7.8% increase over GTX480). To alleviate the off-chip bandwidth bottleneck and reduce memory access latency, GPUs are equipped with a multiple-level on-chip memory hierarchy including register files, L1 data caches (D-cache), shared memory, and L2 caches. As expected, how to effectively utilize such on-chip memory resources has a significant impact on application performance. However, it is non-trivial for application developers to explicitly manage these on-chip memory resources as the trade-offs among these resources are complex and sometimes non-intuitive [14]. More importantly, as on-chip resources have been changing significantly for different generations of GPUs, an optimized kernel upon one generation becomes suboptimal on another one. Thus performance portability is a daunting challenge for application developers.

In this paper, we propose compiler-driven automatic data placement as our solution. We focus on GPGPU programs that have already been reasonably optimized either manually by programmers or automatically by some compiler tools. In other words, our input programs already employ classical loop optimizations such as tiling and allocate important data, either for communication among threads or for data reuses, in shared memory. Our proposed compiler algorithm refines these programs by revising data placement across different types of GPU on-chip memory resources.

Our compiler algorithm places data into different types of on-chip memory resources using the following systematic way. First, it analyzes the usage patterns of all shared memory variables in an input kernel program and tries to promote those shared memory variables into registers if they are not used for communication among threads. Second, if the shared memory usage becomes the bottleneck for thread-level parallelism (TLP), it checks whether it is profitable to move some shared memory variables into either global or local memory so as to implicitly exploit the L1 D-cache. Third, it detects redundant accesses to both global memory and shared memory across different threads. Then, it aims to reduce such redundant accesses by compacting multiple threads into one, thus converting redundant shared/global memory accesses among threads into data sharing/reuse of registers. To find the most profitable data (re)placements, an auto-

2015 IEEE/ACM International Symposium on Code Generation and Optimization
978-1-4799-8161-8/15/$31.00 ©2015 IEEE

tuning process is used to select the optimal parameters in the optimization process. The first two steps of our compiler algorithm focus on replacing shared memory variables with registers or global/local memory variables. The key reason is due to the evolution trend of GPU on-chip memory resources. In early generations such as the Nvidia G80 and GT200 architecture, the ratio of the register file size to the shared memory size is 2 and 4, respectively. In comparison, in the FERMI and KEPLER architecture, the ratio becomes 2.7 and 5.3, respectively. As a result, the code optimized for G80 or FERMI tends to over-utilize shared memory while underutilizing the register file when it runs on GT200 or KEPLER GPUs. As a result of such underutilization, it is proposed in prior works [1] to turn off significant portions of the register file to reduce static power consumption.

We evaluate our proposed automatic data replacement algorithm using a diverse set of applications from different GPGPU benchmark suites that have been manually optimized. Our results show that our compiler algorithm improves the performance by up to 4.14X and an average 1.76X on the FERMI architecture, and by up to 3.30X and an average of 1.61X on the KEPLER architecture.

The remainder of the paper is organized as follows. Section 2 presents a brief background on GPU architecture with an emphasis on on-chip memory resources. Section 3 presents in detail our proposed automatic data placement algorithm. Section 4 and 5 discuss our experimental methodology and the experimental results. Section 6 addresses the related work. Section 7 concludes our paper.

2. Background and Motivation

2.1 GPGPU Architecture and Programming Model

State-of-art GPUs employs many-core architecture, on which the cores are organized in a two-level hierarchy. Each GPU contains a cluster of streaming multiprocessors (SM) in Nvidia GPUs, or computing units in AMD GPUs. Each SM in turn consists of multiple streaming processors (SPs). To amortize the overhead of instruction fetch and decode, an array of SPs executes one scalar program in the single-instruction multiple-data (SIMD) manner. A group of threads running on such an array of SPs and sharing the same program counter (PC) is referred to as a warp of threads. Multiple warps of threads are grouped into a thread block (TB) and a number of thread blocks are organized into a thread grid.

2.2 GPU Memory Resources

The GPU off-chip memory space consists of texture memory, constant memory, local memory, and global memory. Texture memory and constant memory are for read-only data which can be accessed by all threads. Global memory can be read or written by all threads in a kernel. In contrast, local memory is private to each thread.

In order to reduce the latency and improve the bandwidth of off-chip memory accesses, three types of on-chip memory including shared memory, data caches, and a register file, are introduced in each SM. Texture caches and constant caches are also on-chip memory but they are used for read-only data and not our focus in this study.

Among three types of on-chip memory, the register file has the shortest access latency and highest throughput. Furthermore, the register file is larger than the L1 D-cache and shared memory as shown in Table 1. The register file is private to each thread, which means data in registers can only be accessed by a single thread, except for the latest Nvidia KEPLER architecture, which introduces a new instruction "__shfl" [18] to enable a thread to access the registers in other threads within the same warp. The maximum number of registers per thread is ISA-dependent and varies in different architectures. Exceedingly heavy usage of registers per thread will result in register spills into its local memory, which may be captured in L1 D-cache.

Compared to register files, shared memory has lower throughput and smaller capacity. As shown in Table 1, a GTX 680 GPU has a 256KB register file and 48KB shared memory. As shared memory is accessible to all threads in a TB and has low access latency, prior works have been focused on using shared memory to achieve memory coalescing, to provide data communication, and to store data for temporal reuses. L1 D-cache shares the same hardware resource as shared memory on FERMI or KEPLER architecture, In contrast to shared memory, which is explicitly managed by kernel code, L1 D-caches are hidden from developers and are implicitly managed by hardware to keep the most recently accessed data. Furthermore, while the intensive usage of shard memory or registers can limit the number of threads running on each SM, the usage of L1 D-cache does not. However, too many threads in a SM would compete with each other for the limited L1 D-cache capacity, which may result in poor performance due to cache contention [10].

2.3 Architecture Evolutions

GPUs evolve at a fast pace. Taking Nvidia GPUs as an example, from the first generations of unified shader G80 to the state-of-art KEPLER architecture. A comparison of them is shown in Table 1. Several observations can be made from the table. First, there is a higher increase in computational throughput than off-chip memory bandwidth. For example, from the FERMI architecture to the KEPLER architecture, the computation throughput increases by up to 229% while the memory bandwidth increases by only 8.3%. As a result, we need to more carefully manage on-chip resource to effectively utilize the computational resources. Second, among GPU on-chip memory resources, the register file size and D-cache/shared memory have been changing across different generations.

For example, From G80 to GT200, the register file size is doubled while the shared memory capacity remains the same. The same trend is present when comparing the FERMI architecture and the KEPLER architecture. Consequently, the code optimized for early GPU generations tend to use shared memory more heavily. This leads to a serious challenge for performance portability for such optimized code running on different GPUs.

Table 1. A comparison of hardware characteristics across different GPU generations.

	G80 (GTX 8800)	GT200 (GTX 280)	FERMI (GTX 480)	KEPLER (GTX 680)	KEPLER (K20c)
Arithmetic throughput (Gflops/S)	504	933	1345	3090	3950
Memory Bandwidth (GB/S)	57	141	177	192	250
Shared memory size(KB)	16	16	48	48	48
Register file size(KB)	32	64	128	256	256

In summary, the main challenges for application developers to manually manage the on-chip memory resources include: 1) GPUs have three types on-chip memory and, although critical to performance, it is difficult to decide the proper on-chip resource for a particular data element in an application, and 2) the resource evolution is not linear across different GPU generations, and optimal on-chip resource usage varies for different GPU generations.

3. Automatic Data Placement into On-chip Memory Resources

To automatically manage on-chip memory resources and achieve performance portability, in this section, we describe in detail our proposed compiler algorithm for automatic data placement. We first present our analysis of possible data placement patterns among different types of on-chip memory resources. Then, we construct our compiler algorithm using the profitable patterns.

3.1 Data Placement Patterns

As discussed in Section 2, we focus on three types of on-chip memory: register files, shared memory, and L1 D-caches. We propose to move data from one type of on-chip memory to another to achieve optimal resource utilization. As shown in Figure 1, there are six possible directions of moving data variables or six ways of data (re)placement. Data placement between register variables and local memory variables, i.e., direction 3 and 6, is determined by the compiler of the GPU vendors. With the Nvidia GPU compiler NVCC [3], we determine that the array variables accessed with non-constant indices, e.g., $A[k]$ where k is a run-time variable, are allocated in local memory. Both scalars and array variables with constant indices are candidates for register allocation. Moving data from register files and D-caches (i.e., local/global variables) into

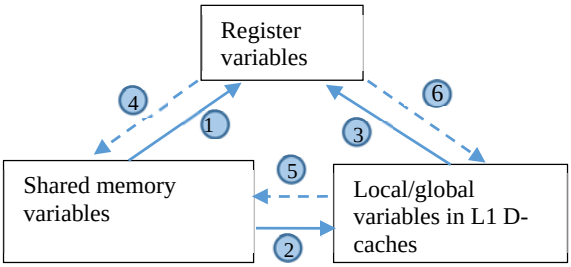


Figure 1. Data placements among three types of on-chip memory.

shared memory, i.e., direction 4 and 5, requires significant code changes besides synchronization. Also, the current trend of GPU evolution is that the register files are much larger than shared memory and the existing compiler tools already can make use of shared memory for data reuse and communication. Therefore, we focus on placement 1, 2, and 3, and leave further investigation on placement 4 and 5 as future work.

3.1.1 Pattern 1: Promote variables from shared memory to registers

Shared memory can be used to exchange data among threads in a TB. Also, as a low-latency on-chip resource, many applications use shared memory as software-managed cache to hold important (private) data for each thread. There are three reasons why it may be profitable to promote a shared memory variable into registers. First, the shared memory usage may limit the number of concurrent TBs on an SM, i.e., TLP, and promoting shared memory variables into registers can alleviate the pressure on this critical resource. Second, shared memory has longer access latency and lower bandwidth than register files. Third, accessing shared memory variables is associated with instruction overhead for address computations. Therefore, higher performance may be expected when promoting

```
1  __global__ void dynproc_kernel(…){
2  __shared__ float prev[256];
3  __shared__ float result [256];
4  int tx=threadIdx.x ;
5  for (int i=0; i<iteration ; i++){
6      …. shortest = minum( prev[tx-1], prev[tx],prev[tx+1]);
7    result[tx] = shortest + gpuWall[index];  __syncthreads();
8    prev[tx]= result[tx];  __syncthreads();}
9    gpuResults[xidx]=result[tx];
10 }
```

a) Baseline

```
1  __global__ void dynproc_kernel(…){
2  __shared__ float prev[256];
3  float result;
4  int tx=threadIdx.x ;
5  for (int i=0; i<iteration ; i++){
6      … shortest = minum( prev[tx-1], prev[tx],prev[tx+1]);
7    result = shortest + gpuWall[index]; ~~syncthreads();~~
8    prev[tx]= result; __syncthreads();}
9    gpuResults[xidx]=result;
10 }
```

b) Optimized Code

Figure 2. A code example of PathFinder.

shared memory variables into registers.

We show a benchmark, PathFinder, as an example, in Figure 2. Path-Finder makes use of two shared memory arrays, '*prev*' and '*result*', as shown in Figure 2. 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 actually 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 indeed shared among different threads. As shown in line 7 in Figure 2a, the '*result*' array is accessed by each thread multiple times in a loop. As each thread only accesses the array result using its own thread id as shown in line 8 and line 9 in the figure, there is no communication using the '*result*' array across threads. Since each thread only accesses its individual part of the array, it is safe to simply replace '*result*[*tx*]' with a register. Further, as the variable is only defined and used in the same thread, we can safely remove the synchronization instruction '__syncthread()' after the statement updating the variable '*result*' (line 7). The resulting code is shown in Figure 2b.

In our study, we found that shared memory is used very often in many benchmarks. Therefore, there are usually multiple shared memory arrays that can be replaced with registers. In this case, we may not have enough registers to promote all the shared memory arrays, and need to decide which shared memory array should be replaced with registers to maximize the performance benefits. Our framework handles this problem by counting the references of each shared memory array, and gives higher priority to the one with larger reference counts (Section 3.3).

3.1.2 Pattern 2: Promote variables from shared memory into L1 D-caches

As discussed above, the register file cannot be used for an array variable with a dynamically determined index (e.g., *A*[*x*]) and intensive usage of registers for shared memory promotion can also limit TLP. The local memory or global memory, which implicitly utilizes the L1 D-cache to achieve the high performance, does not have such drawbacks. Therefore, promoting variables from shared memory into local memory / global memory (L1 D-cache) is a better choice when (1) replacing shared memory arrays with dynamic indices or (2) the shared memory array to be promoted has a large size (e.g., an array of structures). Furthermore, if a shared memory variable is used for communication among threads, a global memory variable can be used to replace it since global memory is visible for all threads.

Figure 3a, using the benchmark, Marching-Cube (MC), from CUDA SDK [19] as an example, shows that two shared memory arrays '*vertlist*' and '*normlist*' are used in

```
1  __global__ void generateTriangles(…) {
2  __shared__ float3 vertlist[12*NTHREADS]; //NTHREADS = 32
3  __shared__ float3 normlist[12*NTHREADS];
4    //defines to the shared memory array
5   vertexInterp2(..., vertlist[threadIdx.x], normlist[threadIdx.x]));
6   vertexInterp2(…,vertlist[threadIdx.x+NTHREADS],
7        normlist[threadIdx.x+NTHREADS]);
8   …edge = tech1Dfetch(triTex,..);
9   //uses of the shared memory array
10  pos[index] =
11    make_float4(vertlist[(edge*NTHREADS)+threadIdx.x], 1.0f);
12    …
13 }
```

(a)

```
1  __global__ void generateTriangles(…) {
2   float3 vertlist[12];
3   float3 normlist[12];
4   //defines to the local memory array
5   vertexInterp2(.., vertlist[0], normlist[0]);
6   vertexInterp2(…,vertlist[1], normlist[1]);
7   … edge = tech1Dfetch(triTex,..);
8   //uses of the local memory array
9   pos[index] =
10    make_float4 (vertlist[edge], 1.0f);
11  …
12 }
```

(b)

Figure 3. A code segment of the benchmark Matching Cube (MC). (a) The shared-memory version, (b) the L1 D-cache version.

the kernel. Each thread only accesses part of these two arrays, and the total size of these two arrays is 9216 bytes for each TB. As a result, each SM can run 5 TBs concurrently even when the shared memory is configured to be 48KB. As we can see from the figure, the value of variable '*edge*' in line 11 of Figure 3a can only be determined in the runtime, and therefore the array '*vertlist*' cannot be allocated in the registers. We choose to promote these two arrays into local memory instead of global memory to minimize the code change since for global memory, we have to modify the CPU code to allocate a global memory array and insert it as a parameter of the kernel invocation. The resulting code is shown in Figure 3b. Since the code in Figure 3b does not use shared memory any more, each SM can run up to 16 TBs in the KEPLER GPUs and 8 TBs in the FERMI GPUs. Such improved TLP leads to higher performance for MC. In many cases, an application may intensively use shared memory to communicate among threads. Then, the global memory has to be used to replace the shared memory variables to maintain such communication so that we can both overcome the TLP bottleneck imposed by shared memory usage and keep inter-thread data communication. Note that although promoting variables from shared memory into L1 D-cache can significantly improve the TLP (or occupancy) otherwise limited by shared memory capacity, it doesn't mean that more TLP will always lead to higher performance. In some scenarios, more concurrent TBs may increase cache and/or network contentions and adversely affect the performance [10]. Thus, our compiler

```
1  __global__ void srad_kernel(int [] c_cuda…){
2   int index_s = cols * BLOCK_SIZE * by + BLOCK_SIZE * bx
3     + cols * BLOCK_SIZE + tx; //BLOCK_SIZE = 16;
4   __shared__ float south_c[BLOCK_SIZE][BLOCK_SIZE];
5   ….
6   south_c[ty][tx] = c_cuda[index_s]
7   if ( by == gridDim.y - 1 ){
8      south_c[ty][tx] = c_cuda[cols * BLOCK_SIZE *
9         (gridDim.y - 1) + BLOCK_SIZE * bx +
10        cols * ( BLOCK_SIZE - 1 ) + tx];
11  }
12  __syncthreads();
13  …}
```

(a)

```
1  __global__ void srad_kernel(int [] c_cuda…){
2   int index_s = cols * BLOCK_SIZE * by + BLOCK_SIZE * bx
3     + cols * BLOCK_SIZE + tx;  //BLOCK_SIZE = 16;
4   __shared__ float south_c[BLOCK_SIZE][BLOCK_SIZE];
5   ….
6   int tmp_1= c_cuda[index_s];
7   //if(by == gridDim.y – 1)
8   tmp_2= c_cuda[cols *  BLOCK_SIZE * (gridDim.y - 1)
9    + BLOCK_SIZE * bx + cols * ( BLOCK_SIZE - 1 ) + tx];
10  #pragma unroll
11  for(int m=0;m<C_Factor; m++)
12   south_c[ty+ m*blockDim.y/C_Factor][tx] = tmp_1;
13   if ( by == gridDim.y - 1 ){
14    for(int m=0;m<C_Factor; m++)
15     south_c[ty+m*blockDim.y/C_Factor][tx] =tmp_2;
16  }
17  __syncthreads();
18 …}
```

(b)

Figure 4. A code segment of the benchmark SRAD. (a) The global memory version, (b) The register version.

uses an auto-tuning process to determine (1) how many variables should be promoted and (2) whether they are promoted into local memory or global memory, so as to achieve optimal data placement in balancing trade-offs between TLP and network/memory pressure.

3.1.3 Pattern 3: Promote variables from shared memory / global memory into registers to achieve register tiling

A common side effect of single-program multiple-data (SPMD) parallelization is redundant computations and memory accesses. In GPU kernels, there often exist redundant accesses to either shared memory data or global memory data across different threads. This redundant shared/global memory reference can be promoted into register usage to further save bandwidth.

We use the benchmark SRAD as an example to illustrate this behaviour. Figure 4a shows a code segment from the SRAD kernel code. The TB dimension of the SRAD kernel is configured as <16,16>, i.e., 256 threads per TB. Therefore, *tx* (i.e., *threadIdx.x*) ranges from 0 to 15 for all the warps in a TB; and *ty* (i.e., *threadIdx.y*) will be 0~1 for the first warp, 2~3 for the second warp, and so on. Before computation, a tile of data will be loaded from the global memory array '*c_cuda*' into the shared memory array '*south_c*' as shown in line 8 of Figure 4a. We can see that the index variable '*index_s*' is dependent on *tx*, *bx* (i.e., *blockIdx.x*) and *by* (i.e., *blockIdx.y*) but not on *ty*. It means that when the 8 warps of a TB actually load the same block of global memory data, there are 7 redundant global memory accesses in each TB since all the warps share the same *tx*, *bx*, and *by*, i.e., the same memory reference index.

All three types of on-chip memory can be potentially used to reduce the overhead of such redundant global memory accesses across warps. First, the L1 D-cache is utilized implicitly when redundant global memory accesses hit in the L1 D-cache but such data reuse cannot be assured as the data may be evicted by other data requests. Second, we can choose to let only the first warp load the data into shared memory, and other warps then access the data from shared memory. However, this way incurs overhead due to operations moving data from/into register into/from shared memory [14]. Additional control flow is also needed to ensure that the global memory data are loaded only once and a synchronization is necessary to eliminate potential data races. Third, although the register file has a large size and the lowest latency, it cannot be shared among warps. In order to take advantage of the register file, we need to first compact multiple warps/threads into a single warp/thread, and then promote shared/global memory variables into registers. This way, the register variables after thread compaction can be shared among different threads before compaction. Such thread compaction is also referred to as thread merge [26] and thread coarsening [11]. Compared to the prior works on thread merge/coarsening/fusion [26][15][22], our approach specifically leverages this optimization technique for register tiling, i.e., use register reuse to eliminate redundant shared/global memory accesses. A key question for such register tiling is how many threads to be compacted so as to maximize register reuse while restricting the register pressure on TLP. We introduce the compaction factor *C_Factor* in our compiler algorithm to determine the most profitable version of data placement using automatic tuning.

The optimized code after compaction is shown in Figure 4b. The number of original threads/warps to be compacted is defined as a run-time constant, *C_Factor*. First, the thread block dimension is adjusted from <16,16> to <16,16/*C_Factor*>. Second, the global memory read accesses on line 6 of Figure 4a are replaced with a single global memory access on line 6 of Figure 4b, which loads the data from global memory to the register variable '*tmp_1*'. Third, since multiple threads/warps of Figure 4a are compacted into a single thread/warp in Figure 4b, we can reuse the register '*tmp_1*' as shown in line 11. Similarly, the memory access of '*c_cuda*' under the conditional statement (line 8 of Figure 4a) can be processed in the same way by introducing another register '*tmp_2*' as shown in Figure 4b. The if statement in line 7 of Figure 4a sometimes may also need to be replicated to guard this '*c_cuda*' access to avoid potential out-of bound accesses.

3.2 Compiler Algorithms and Implementation

Although the data placement patterns discussed in Section 3.1 can be used to guide programmers to manually optimize their GPU programs, it will quickly become un-manageable if a non-trivial number of data variables are to be analyzed. In this section, we present our source-to-source compiler framework which implements these three data placement patterns using an automatic compiler optimization algorithm. The goal of the compiler algorithm is to generate the code which utilizes on-chip resource efficiently without effort from application developers. The key feature is that the compiler framework can intelligently re-assign the memory types of variables of a GPU program to maximize the benefit of on-chip resources. Our compiler algorithm has two passes: one for data placement pattern 1 and pattern 2 and the other one for data placement pattern 3.

Either compiler pass has three stages: the identifying stage, the processing stage, and the auto-tuning stage, as detailed in Figures 5 and 6. The identifying stage will scan all the memory variables, and generate a list of candidate variables which can be promoted by collecting the architecture features and analyzing the memory accesses of the target kernel. The processing stage implements the data placement patterns by revising the data types and their access indices of these candidate variables. The auto-tuning stage constructs the search spaces, decides which variables to be processed and selects the optimal code versions.

3.2.1 Compiler pass 1

The algorithm of the compiler pass for promoting shared memory variables to register files/local memory/global memory is shown in Figure 5. The identifying stage (line 5~15) collects all shared memory variables through their '*__shared__*' keyword. For shared memory variables, we mark an access as a combination of the array name and the access index. The compiler checks access indices to determine (a) whether an access is across different threads or private to a single thread, and (b) whether an index has to be determined at the runtime. Meanwhile, the reference count of the variable is also recorded. If an access is inside a loop, we weight this access number by timing a loop count in line 10. In some cases, the loop count in a one-level loop or multiple loop counts in nested loops may associate with a run-time value, leading to some unknown reference counts. In such cases, we resort to either profiling or simple heuristics (Section 3.2.4). The output of the identifying stage is *arrays*, a list of candidate variables.

For all the candidate variables in *arrays*, the processing stage (line 18 ~24) applies data placement patterns by first selecting the shared memory variable with the largest reference counts. Then, if a shared memory variable is not shared across threads and is not accessed with run-time determined indices, it is promoted to the register file. Otherwise, it is replaced with a global memory variable if

```
1  Kernel shared_to_register_or_local_or_global (Kernel kernel) {
2    Kernel best_kernel = kernel;
3    float exe_time = eval(kernel); //collect the execution time of kernel;
4    /**Identification Stage**/
5    List arrays;
6    for (each shared memory array sma in kernel) {
7        sma.is_overlap = false; sma.is_index = false;
8        sma.access_count = 0;  sma.size = allocation_size;
9        for (each access acc of array sma) {
10         sma.access_count += (acc in loop)?loop_count::1;
11         if (acc is overlapped across threads)
12           sam.is_overlap = true;
13         else if (the address of acc is calculated in the runtime)
14           sma.is_index = true ;}
15     if (sma.access_count >0) {arrays.add(sma);} }   //end for
16   while (arrays is not usage empty)  {
17     /**Processing Stage**/
18     sma = array with largest access_count in arrays, pop it out;
19     if (!sma.is_index and !sma.is_overlap)
20         replace sma with register file;
21     else  if (sma.is_index and !sma.is_overlap)
22         replace sma with local memory;
23     else
24         replace sma with global memory;
25     /**Auto-tuning Stage**/
26     generate a new kernel nkernel
27     exe_time1=eval(nkernel) //the execution time of nkernel
28     if (exe_time1< exe_time) { // the new kernel is better
29         best_kernel = nkernel;
30         exe_time = exe_time ;}
31     else
32         return best_kernel; // found the best kernel  }   //end while
33 }
```

Figure 5. The compiler algorithm to promote variables from shared memory to the register file/D-cache.

is used for inter-thread communication in line 23~24; or it will be replaced with a local memory variable if it is accessed through indices at line 21~22. Each replacement will result in substituting both the variable definition and reference indices throughout the kernel code from original one to the promoted type.

3.2.2 Compiler pass 2

The second compiler pass implements the third data placement pattern, i.e. promoting redundant shared/global memory accesses into register accesses, as shown in Figure 6. In the identifying stage (line 5~13), the compiler analyzes each shared or global memory array. It checks whether an array index is independent upon the thread id in either the *X* or *Y* dimension. If it is independent upon both dimensions, it sets the flag *is_redundant_2d*. Otherwise, if it is independent upon one direction, it sets the flag *is_redundant_1d*. During each index check, the compiler also inserts the expressions associated with the index into the *exprs* list, which will be used in the processing stage. After the identification stage, it outputs *exprs*, the list of candidate expressions that exhibit data access redundancy, and the corresponding flags that indicates the type of redundancy type, i.e., one-dimension or two-dimension.

In the processing stage (line 16~27), the compiler first adjusts the thread block dimension for each different

```
1  Kernel shared_or_global_access_to_register (Kernel kernel) {
2    Kernel best_kernel = kernel;
3    float exe_time = eval(kernel);
4    /**Identification Stage**/
5    List exprs;
6    bool is_redundant_1d = false, is_redundant_2d = false;
7    for (each shared/global memory array sma in kernel) {
8        for (each access acc of array sma in expression expr) {
9            if (acc is independent of one thread dimension)
10               { is_redundant_1d = true; exprs.add (expr);}
11           if (is_redundant_1d && acc is independent of the other
12               thread dimension in expression expr)
13               {is_redundant_2d = true; exprs.add (expr);} } }//end for
14   for (each C_Factor in search spaces ){
15       /**Processing Stage**/
16       Adjust Thread Block Dimension.
17       if(is_redundant_1d) {
18           construct a one-loop with loop bound C_Factor to
19               perform the workload for compacted threads
20           convert expr in exprs to from inter-thread memory usage
21               into register array.
22       } else if(is_redundant_2d){
23           construct an 2-level loop with loop bound C_Factor .x,
24              and C_Factor . y to perform the workload
25              for compacted threads
26           convert expr in exprs to from inter-thread memory
27               usage into register array usage; }
28       /**Auto-tuning Stage**/
29       generate a new kernel nkernel from best_kernel;
30       exe_time1=eval(nkernel) //the execution time of nkernel
31       if (exe_time1< exe_time) {   // the new kernel is better
32           best_kernel = nkernel;
33           exe_time = exe_time; }
34       else
35           return best_kernel;    } // end for
36   }
```

Figure 6. The compiler algorithm to promote shared or global memory to registers to be shared among threads.

compact factor (*C_Factor*) in line 16. Then, it constructs an unroll-able loop to perform the workloads of the threads to be compacted. The loop body contains all the expressions including the associated computational operations and memory accesses in the thread index dependence chain, as shown in line 18~19 and line 23~25. The exception is the expressions in the *expr* lists collected in the identification stage. These global/shared memory accesses in the *expr* list are performed only once by loading data into a destination register as shown in line 20~21 and 26~27 and the destination register will be reused in the newly constructed loop, as illustrated in Figure 4. This *C_Factor* is a tunable parameter to indicate how many threads/warps will be compacted into one. The *C_Factor* can be a scalar or a two-dimension vector depending on the redundancy type generated from the identifying stage.

3.2.3 Auto-tuning

In the auto-tuning stage, the compiler first generates a search space based on all the tunable parameters, then measures the execution time, compares them, and finally selects the best performing version. Totally, three search spaces will be generated associated with each data placement policy. The first search space is to decide how many and which shared memory variables to be promoted into the register file; the second search space is which shared memory variables to be promoted into global/local memory; the third search space is to determine the compaction factor.

To manage the cost of auto-tuning, we prune the first search space by promoting shared memory variables incrementally, starting from the one with the highest reference counts, assuming that this one will benefit most when being promoted into registers. If one version has lower performance than the previous one, it means that promoting one more shared memory variable may lead to too much register usage and hurt the performance. Therefore, it stops further promotions. For the second search space, we prune it by using a greedy strategy to promote shared memory variables that occupy the largest space, so that it will release the resource pressure on TLP in a fast and incremental way. For the compaction factor, we observe that, the thread block in GPU computing workloads is typically a multiple of 32. Therefore, we constrain the compaction factor as a number of 2's power. Compared to the previous sophisticated methods for pruning the search space such as generic algorithm [5] and machine learning techniques [20], our heuristics are simple and practical on GPU kernels. In Section 5, our results also show that for our workloads, our iterative space pruning approach is effective in reducing search space and finding the optimal/near optimal version.

The auto-tuning part for compiler pass 1 is listed as line 27 to line 32 in Figure 5. During auto-tuning, if a newly generated kernel has worse performance than the previous version, the compiler will consider further optimization is not helpful and the previous version is chosen as the best one as shown in line 32. In Figure 6, the code lines from 29 to 35 show the auto-tuning part for compiler pass 2. The compiler evaluates the new kernel generated by previous steps in line 30. If the new kernel has better performance, then the compiler increases the C_Factor and continues with more aggressive thread compaction. Otherwise, the compiler stops at line 35.

3.2.4 Preprocessor

In our implementation, our compiler framework takes a pre-processing step on the program and regulates expression representation for successive analysis and optimization. First, the index for an array access is interpreted as an affine function of the thread index. The scaling factor in the affine function may involve a subset of the kernel launch parameters, macro/constant values, run-time parameters, and loop iterators if the memory expression is inside a loop. Second, an array access may reside inside a condition or loop statement. The reference count of such an array depends on the loop bound and the

condition. If the loop bound and the condition can only be determined at run-time, we choose to either let the user to provide such information through profiling or use the following simple heuristics. We assume that for a nested loop in a kernel, each level has a loop count of 4 and the condition is true half of the times. The reason for such a default loop count is that our observation from the benchmarks shows that when a nested loop is parallelized into GPU threads, the levels with large loops counts are used to generate thread grids and the thread body typically contains loops with smaller counts. Lastly, the preprocessor collects the data structure declaration and annotate array accesses with the data type. For the vector data type such as int2, float4, the memory access index is processed the same as the scalar data type. For the struct type, the array index and the addresses of its elements are identified separately.

4. Experimental Methodology

We implemented our compiler algorithms using Cetus [13], a source-to-source compiler infrastructure for C programs. The CUDA syntax support is ported from MCUDA [21].

Table 2. Parameters used in experiments.

Parameter	GTX480	GTX680	K20c
<Shared memory size, L1 D-cache size>	<16kB, 48kB>, <48kB, 16kB>	<16kB, 48kB>, <32kB, 32kB>, <48kB, 16kB>	<16kB, 48kB>, <32kB, 32kB>, <48kB, 16kB>
Register file size	128kB	256kB	256kB
Max number of threads per SM	512	1024	1536
Max number of registers per thread	64	64	256
Compaction Factor	2,4,8,16	2,4,8,16	2,4,8,16

To evaluate our proposed compiler optimizations, we perform our experiments on Nvidia GTX480 (FERMI) GPUs, GTX680 (KEPLER) GPUs, and Telsa K20C GPUs. The parameters are presented in Table 2.

Most of the benchmark kernels used in our experiments are from Rodinia [4] and CUDA SDK [19] since they have already been manually optimized. Among them, HotSpot, Back Propagation, SRAD, Pathfinder, B+tree, LU Decomposition are from the latest Rodinia suite. Matrix Multiplication and MarchingCubes are from CUDA SDK. NQU is from GPGPUsim benchmark suite [2]. As Back Propagation, SRAD and B+tree, contain two GPU kernels, we use BackPropagation1, BackPropagation2, SRAD1, SRAD2, B+tree1, B+tree2 to differentiate them. In Table 3, from left to right, we show the benchmark name, the input, as well as the resource usage including the number of registers per thread and the size of shared memory (bytes) per SM on GTX 480, GTX 680, and Telsa K20c, respectively. We use the default input released with the code. For each benchmark, the shared memory usage is the same for different GPUs because it is determined by programmer's explicit definition. The register usage is statically allocated and the maximum available registers per thread vary on different GPUs.

Table 3. Benchmarks and their resource usage.

		GTX480		GTX680		K20C	
Benchmark	Input	reg	smem	reg	smem	reg	smem
HotSpot (HS)	height 2	35	3072	36	3072	39	3072
Back Prop1 (BP1)	65536 layer	13	1088	11	1088	12	1088
Back Prop 2 (BP2)	65536 layer	22	0	20	0	21	0
SRAD1 (SR1)	2048*2048	20	0	20	0	26	0
SRAD2 (SR2)	2048*2048	19	0	20	0	20	0
Matrix Multiply (MM)	2048*2048	23	8192	26	8192	25	8192
Path Finder (PF)	409600 steps	16	2048	18	2048	17	2048
N-Queue (NQU)	N=8	15	15744	19	15744	16	15744
Marching Cubes (MC)	32768 voxels	63	9216	63	9216	76	9216
B+tree1 (BT1)	qrSize=6000	18	0	19	0	21	0
B+tree2 (BT2)	qrSize=6000	23	0	28	0	30	0
Lu-Decompose (LUD)	2048.dat	15	2048	17	2048	17	2048

5. Experimental Results

In our first experiment, we measure the execution time of both the original kernel and the optimized kernel generated from our compiler algorithm on GTX480, GTX680 and Telsa K20c separately. On each GPU, we tried all different shared memory/L1 D-cache configurations and selected the one with the best performance for the original kernels. Also, for each optimized kernel, the compiler will generate the best data placement to accommodate the specific architecture so as to achieve optimization portability. Each benchmark has been run one-hundred times to obtain the stable execution times. Figure 7 shows performance comparisons between original kernels and our optimized ones across different GPUs.

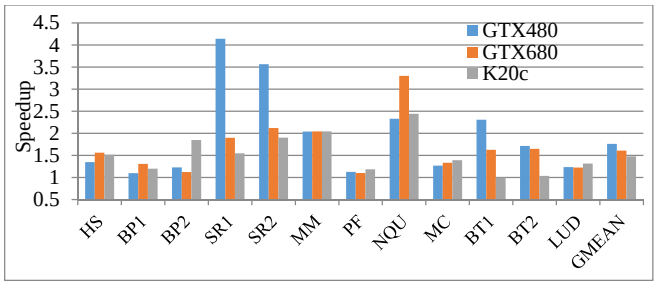
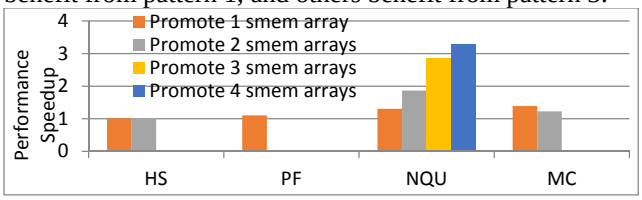


Figure 7. Performance speedups achieved by automatic data placement for all benchmarks on three different GPUs.

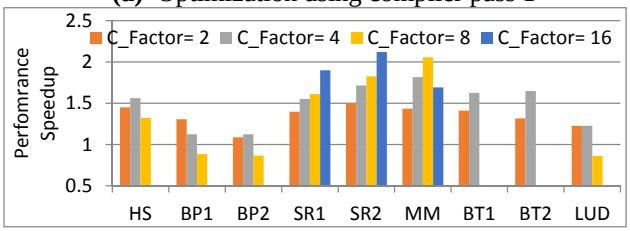
From Figure 7, we can see that across all the benchmarks, the optimized kernels exhibit significantly higher performance than the original ones on all the three GPUs. The benchmark SR1 achieves the highest speedup (4.14X) over the original kernel on GTX480. This is because most global memory accesses in this kernel are redundant, not only among different warps but also among threads in a warp. By promoting those redundant global memory accesses into register accesses, the access time to all input data sets is highly reduced. Similar global or shared memory redundancy also exists in HS, BP1, BP2, MM, BT1, BT2, and LUD. For HS, PF and NQU. Many shared memory variables in these kernels have no data

exchanges among threads, thus these shared memory variables are candidates for register promotion. In NQU, there are five shared memory variables and four of them are promoted, leading to a high performance speed-up of 3.3x on GTX680. For PF, _syncthreads() can be safely removed. However, even though it is not removed, the optimized code (e.g., on GTX480) can still achieve 7% performance improvement. For MC, shared memory variables holding two on-chip working sets can be promoted into local memory arrays so as to remove the resources limitation on the number of concurrent TBs, thereby achieving higher performance. Overall, using the geometric mean as an average, the kernels optimized on GTX480 can achieve up to 4.14x speedup and an average of 1.76x speedup compared to the original benchmarks, up to 3.30x speedup and an average of 1.61x speedup on GTX680, and up to 2.44x speedup and an average of 1.48x speedup on K20c.

In our second experiment, we first breakdown the effectiveness of each placement pattern. Figure 8a and 8b shows the benchmarks that can be applicable to compiler pass 1 and pass 2. Among them, only HS benefits from both pattern 1 and pattern 3 (the total improvement of 64.2%: breakdown into 4.8% from pattern 1 and 59.4% from pattern 3), while other benchmarks only benefit from one in three patterns: MC benefits from pattern 2; PF, NQU benefit from pattern 1, and others benefit from pattern 3.



(a) Optimization using compiler pass 1



(b) Optimization using compiler pass 2

Figure 8. Auto-tuning of our automatic data-placement for all benchmarks on GTX680 (Performance normalized to original kernel).

We further evaluate the effectiveness of our auto-tuning process for each benchmark. As shown in Figure 8a, the benchmarks NQU, PF, HS and MC benefit from promoting shared memory arrays into register/local/global memory. The search space is how many shared memory variables can be promoted into registers or L1 D-cache using our compiler pass 1 in Section 3.2.1. For all the cases, promoting more shared memory variables into registers or L1 D-cache will lead to higher performance. For the benchmark kernels benefiting from reduced redundant shared/global memory accesses, Figure 8b shows the impact of the search parameter *C_Factor* in our compiler pass 2 in Section 3.2.2. From Figure 8b, we can see that the best *C_Factor* varies across different benchmarks. For SR1, the best performing version is achieved when *C_Factor* is 16. However for BP1, the best performing one is obtained when *C_Factor* is 2, and further increasing *C_Factor* to 4 degrades the performance as it reduces the number of active warps in a thread block. Such reduced TLP subsequently degrades the latency hidden ability for off-chip memory accesses, offsetting the profit from reducing redundant accesses. Therefore, auto-tuning is stopped when such a performance drop is observed. We can see that if *C_Factor* is increased to 8 for BP1, the performance will degrade even more. This validates the effectiveness of our auto-tuning policy, which searches *C_Factor* in an incremental manner. The same scenario has also been observed in the compiler pass 1 from Figure 8a when searching for the appropriate shared memory variables to be promoted in MC.

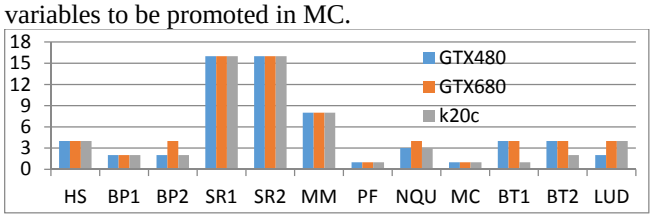


Figure 9. The optimal parameter, the number of shared memory array to be promoted and the C-Factor, determined for different GPUs.

Third, in Figure 9, we present the optimal parameters determined by our auto-tuning process on the different GPUs. For PF, NQU and MC, the y-axis means how many variables should be promoted while for others, the y-axis denote the optimal C_Factor values on different GPUs. Our compiler can intelligently generate the optimized kernel for specific architecture to achieve optimization portability. We can see that the different architecture features of these GPUs lead to different optimal parameters. For example, NQU achieves best performance when its four shared memory variables are promoted on

Table 4. The auto-tuning time on GTX 680

	Original search space	Pruned search space	Auto-tuning time (ms)
HS	48	8	42.873
BP1	16	3	11.361
BP2	16	4	15.755
SR1	16	5	24.133
SR2	16	5	21.941
MM	32	5	210.876
PF	1	1	8.88
NQU	45	12	48.124
MC	9	6	23.986
BT1	3	3	12.183
BT2	3	3	14.343
LUD	16	4	129.531

GTX680, while on K20C, the best performance is achieved when three shared memory variables are promoted.

Fourth, our auto-tuning process has a low overhead on searching the optimal parameters. We report the cost of the auto-tuning function in Table 4. From the left to right, we report the search space, i.e., the number of all possible values to be tried, in the original search if there is no pruning strategy in searching, the search space after applying our pruning strategies in our compiler passes, and the total execution time of our auto-tuning process for generating the optimal kernel for each benchmark. We can see that the search space is reduced significantly by our pruning strategy. We also validated that the optimized one from our pruned space can achieve the same performance as the one from the original search space.

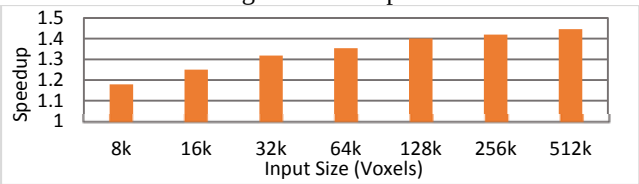


Figure 10. Performance speedup of optimized kernels in Marching Cubes with different input sizes.

Finally, besides the kernel code itself, we also consider how the problem input of a workload affects our proposed optimization process. For our first compiler pass, the shared memory array sizes are fixed with constants or macro variables which are independent of input sizes. The reason is that the benchmark code has been already optimized to process the inputs as tiled working sets. For the second pass, the input size will impact on the number of thread blocks in a grid and each thread block usually has a pre-defined size to work on a tile of input elements. Thus, the variation of the input size will not affect the steps of our compiler analysis and optimizations. Provided that the performance is in general correlated with the input size, our performance improvement will higher when the problem size is larger. Because the larger inputs will often lead to more frequent on-chip memory resource accesses to process them and our optimized kernel will in turn benefit more from the optimized access patterns. Figure 10 shows the effect of increased input size, from 8K Voxels to 512K Voxels, on Marching Cubes. As the input problem size increases, the performance improvement of our optimized kernel from compiler also increases from 1.179x to 1.446x.

6. Related work

In recent years, GPUs have been widely used for general-purpose computation due to their high computational throughput. However, achieving high performance on GPUs is not easy, and one of reasons is the intricate on-chip memory resources. Among on-chip resources, shared memory is controlled by users, and many highly optimized applications or algorithms on GPUs utilize shared memory carefully [12][23][24][27] so as to enjoy the low-access access latency and high bandwidth. Besides them, [12] analyses the upper performance bound of SGEMM on GPUs and optimizes the kernel through register blocking by reusing data in registers as much as possible for maximal throughput. However, none of these works considers the overhead of intensive usage of shared memory and the impacts of varying on-chip resources across different GPU generations.

To relieve the burden of optimizing GPU programs from the programmers, many auto-tuning frameworks [15][16][22][25][26] have been developed to automatically optimize the GPU programs to achieve high performance. For example, a polyhedral model is used in [16] for optimizing global memory accesses. In [27], the shared memory is time multiplexed to reduce the pressure on limited shared memory capacity. In [25], language and compiler support are proposed to leverage nested parallelism inside the GPU programs. However, most of these works focuses on optimizing memory accesses and managing thread-level parallelism using compiler techniques. Management of different types of on-chip memory, especially the varying on-chip memory across different GPU generations, has not been the focus. To the best of our knowledge, our work is the first compiler algorithm to automatically optimize data placement across different on-chip memory resources in a systematic way.

We also observed that vendor's compiler may promote the variables in shared memory to register file. The way to avoid such an optimization is to use the '*__volatile__*' keyword when declaring a shared memory array. However, as we verified from the assembly codes, we found that the vendor's compiler does not apply such optimizations on the benchmarks used in our work.

Current studies on on-chip memory resources mainly focus on identifying resources limitation and boosting the performance by improving architecture design [6][7] or compiler support [9][27]. On-chip data cache may lead to cache contention and [9] proposes a compiler algorithm to automatically turn on/off the D-cache by predicting how cache will affect the performance. The register usage pattern is studied in [6] and the register file accesses are reduced by proposing a register file cache. However, these works target on optimizing one specific resource to conquer their limitations instead of balancing on-chip resources.

The trade-offs between software-managed shared memory and hardware-managed D-cache on GPUs have been studied in [14]. Gebhart et al. [7] made the observation that different applications have different needs for various memory resources. They proposed unified local memory that can dynamically change the partition among registers, cache, and shared memory according to each application's needs. Hayes and Zhang [8] proposed unified on-chip memory allocation which uses shared memory to

offload register pressure. In comparison, our work focuses on re-assigning data across all on-chip memory resources.

7. Conclusions

Judicious utilization of the on-chip memory resources has a significant impact on application performance. However, how to manage these intricate on-chip memory resources is non-trivial for application developers. More importantly, the varying on-chip resource across different GPU generations makes performance portability a daunting challenge. In this paper, we propose compiler-driven automatic data placement as our solution. We focus on GPGPU programs that have already been reasonably optimized either manually by programmers or automatically by existing compiler tools. Our proposed compiler algorithms refine these programs by altering data placement among different on-chip resources to achieve both performance enhancement and performance portability. In particular, we leverage three data placement patterns. First, we explore shared memory variables to promote them into registers. Second, we explore the opportunities to utilize the L1 D-cache by promoting variables from shared memory into global/local memory if shared memory is a resource bottleneck. Third, we eliminate redundant shared/global memory accesses across different threads. To achieve performance portability, our compiler performs auto-tuning on different GPUs to achieve optimal performance. Among the benchmarks in our study, our proposed compiler algorithms significantly improve the performance by up to 4.14x and 1.76x on average on Nvidia GTX480 (i.e., FERMI) GPUs, and by up to 3.30x and 1.61x on average on GTX680 (i.e. KEPLER) GPUs, and up to 2.44x speedup and an average of 1.48x speedup on K20c GPUs. Our compiler-optimized kernel can also save up to 74.3% energy and save an average of 40.3% energy overall measured on GTX680 GPUs.

ACKNOWLEDGMENT

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

References

[1] M. Abdel-Majeed. and M. Annavaram. Warped register file: A power efficient register file for GPGPUs. *HPCA*, 2013.

[2] A. Bakhoda, G. Yuan, W.L. Fung, H. Wong and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. *ISPASS*, 2009.

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

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

[5] K. D. Cooper, P. Schielke and D. Subramanian. Optimzing for reduced code space using generic algorithms. *LCTES*, 1999.

[6] M. Gebhart, et al. Energy-efficient mechanisms for managing thread context in throughput processors. *ISCA*, 2012.

[7] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky and W. J. Dally. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. *MICRO*, 2012.

[8] A. Hayes and E. Zhang, Unified On-Chip Mmeory Allocation for SIMT Architecture, *ICS*, 2014.

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

[10] O. Kayran, A. Jog, M. T. Kandemir and C.R. Das. Neither more nor less: optimizing thread-level parallelism for GPGPUs. *PACT*, 2013.

[11] D. B. Kirk and W. W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach, 2010.

[12] J. Lai and A. Seznec. Performance upper bound analysis and optimization of SGEMM on FERMI and KEPLER GPUs. *CGO*, 2013.

[13] S. Lee, et al. Cetus-An extensible compiler infrastructure for source-to-source transformation. *LCPC*, 2004.

[14] C. Li, Y. Yang, H. Dai, S. Yan, F. Muller and H. Zhou. Understanding the Tradeoffs between Software-Managed vs. Hardware-Managed Caches in GPUs. *ISPASS*, 2014.

[15] Z. Lin, X. Gao, H. Wan and B. Jiang. GLES: A Practical GPGPU Optimizing Compiler Using Data Sharing and Thread Coarsening. *LCPC*, 2014.

[16] M. M. Baskaran, etal. A compiler framework for Optimization of Affine Loop Nests for GPGPUs. *ICS*, 2008.

[17] NVIDIA FERMI: NVIDIA's Next Generation CUDA Compute Architecture, Nov. 2011.

[18] NVIDIA KEPLER GK110 white paper. 2012.

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

[20] M. Stephenson, S. Amarasinghe, M. Martin and U.M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *PLDI*, 2003.

[21] J. A. Stratton, S. S. Stone, and W. W. Hwu, MCUDA: An efficient implementation of CUDA kernels on multi-cores. *LCPC*, 2008.

[22] S. Unkule, C. Shaltz and A.Qasem. Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality. *CC*, 2012.

[23] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X, Shen. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-Coalesced GPU Memory Accesses. *PPoPP*, 2013.

[24] S. Yan, C. Li, Y. Zhang and H.Zhou. yaSpMV: Yet Another SpMV Framework on GPUs. *PPoPP*, 2014.

[25] Y. Yang, and H. Zhou. CUDA-NP: Realizing Nested Thread-Level Parallelism in GPGPU Applications. *PPoPP*, 2014.

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

[27] Y. Yang, P. Xiang, M. Mantor, N. Rubin and H. Zhou. Shared Memory Multiplexing: A Novel Way to Improve GPGPU Performance. PACT, 2012.