Fixing Performance Bugs: An Empirical Study of Open-Source GPGPU Programs

Yi Yang
Dept. of ECE
NCSU
Raleigh, NC, USA
yyang14@ncsu.edu

Ping Xiang
Dept. of ECE
NCSU
Raleigh, NC, USA
pxiang@ncsu.edu

Mike Mantor
Graphics Products Group
AMD
Orlando, FL, USA
Michael.Mantor@amd.com

Huiyang Zhou
Dept. of ECE
NCSU
Raleigh, NC, USA
hzhou@ncsu.edu

***Abstract*— Given the extraordinary computational power of modern graphics processing units (GPUs), general purpose computation on GPUs (GPGPU) has become an increasingly important platform for high performance computing. To better understand how well the GPU resource has been utilized by application developers and then to facilitate them to develop high performance GPGPU code, we conduct an empirical study on GPGPU programs from ten open-source projects. These projects span a wide range of disciplines and many are designed as high performance libraries. Among these projects, we found various performance 'bugs', i.e., code segments leading to inefficient use of GPU hardware. We characterize these performance bugs, and propose the bug fixes. Our experiments confirm both significant performance gains and energy savings from our fixes and reveal interesting insights on different GPUs.**

Keywords: GPGPU; Performance; Compiler;

I. INTRODUCTION

Modern graphics processing units (GPUs) feature both high computational throughput and memory access bandwidth. As a result, general purpose computation on GPUs (GPGPU) or GPU computing has become an increasingly important platform. Although impressive speedups over central processing units (CPUs) have been reported, it remains a challenge to develop high performance GPGPU programs and very often the GPU resources are not utilized efficiently. In order to understand how well application developers use the GPU hardware for their computations and then to facilitate them to develop high performance GPGPU code, we conduct an empirical study on the GPGPU programs from ten open-source projects and present what we learned from inspecting the programs.

To make our results representative, we select these open-source projects from a wide range of disciplines, including computational physics, atmosphere science, bioinformatics, mathematics, machine learning, etc. Such a wide range helps us to prevent the bias based on our own domain of interest. We make sure that our selected projects already achieve reasonably good performance based on the published results or our own tests.

Despite our lack of application domain knowledge, our detailed code inspection reveals that many of the GPGPU programs under our study can be improved to achieve both significantly higher performance and better energy efficiency. We identify the common code patterns that lead to inefficient use of GPU hardware and refer to them as performance bugs. We group the performance bugs into the following categories: (a) global memory data types and access patterns, (b) the thread block dimensions as a kernel invocation parameter, (c) portability across different GPUs, (d) the use of constant and texture memory, (e) function specialization, and (f) floating-point number computations. For all these bugs, current compilers have not been able to fix the adverse performance impact.

The key issue with the global memory data types and access patterns is that the memory coalescing requirements are often violated. Given the prior work on this issue [3][6][14][31], we briefly address this category of bugs and only elaborate the effects of new hardware features such as the level-1 (L1) cache for the global memory.

Thread hierarchy determines how threads are distributed among multiple streaming multiprocessors (or SIMD engines) on a GPU, which in turn decides how global memory data are accessed and reused in either shared memory or hardware caches. We present a detailed study on the effect of thread block dimensions in the thread hierarchy and show that without any modification to the kernel code, the performance of an open-source function can be improved by up to 1.77X by only changing the thread block dimension.

OpenCL (Open Computing Language [18]) is a cross-vendor open platform to provide portability across different GPUs. Although modern GPUs have similar high-level architecture, the GPUs from different vendors (e.g., AMD HD5870 vs. NVIDIA GTX480) or the different generations of GPUs from the same manufacturer (e.g., GTX285 vs. GTX480) have different hardware features. Our study shows that the code optimized for one device typically will not be as efficient in another device and the code developed in OpenCL without consideration of detailed hardware features may fail to achieve high performance for any device. We propose a set of rules to address this performance portability problem.

Besides the global memory, the texture memory and/or the constant memory can deliver very high data bandwidth using the on-chip constant caches and texture caches. Among the GPGPU projects under our study, few use such valuable resources. We show that with effective utilization of the constant memory, our novel matrix multiplication outperforms NVIDIA CUBLAS 3.1 [15] by up to 74% and CUBLAS 3.2 [16] by up to 30% on GTX480.

In some open-source functions, variables of limited range of values are used. Since they are declared as run-time variables instead of static constants, the opportunity for compiler optimizations is reduced. We propose to generate multiple versions of a function; each specialized with a particular constant value. With such function specialization,

we can promote shared memory variables to registers and enable loop unrolling optimizations.

It is well known that GPUs delivers higher throughput on single-precision floating-point (FP) numbers than double-precision ones. However, it is important to know how data types are processed by the compiler in order to avoid un-intentional double-precision computations. Furthermore, we found that when double-precision is required, re-arranging the computations may leverage the specialized support of certain functions in GPU hardware or reduce the number of unnecessary operations.

Given the importance of energy efficiency in high performance computing, we also characterize the energy impact on the performance bugs. Since these bugs are a result of inefficient utilization of GPU hardware resources, fixing them leads to a significant amount of energy savings.

In summary, our work makes the following contributions. (1) We investigate ten open source projects and characterize the common performance bugs issues. (2) We propose a set of new optimization techniques to fix the performance bugs of these open source projects. (3) We study the energy effect of performance bugs and show that our proposed fixes achieve both high performance and energy efficiency.

The remainder of this paper is organized as follows. Section II presents an overview of the selected open-source GPGPU projects. Section III addresses the background on GPU computing and highlights the key to achieve high performance. Experimental methodology is presented in Section IV. The performance bugs and our proposed fixes are discussed in detail in Section V. Section VI discusses the related work and Section VII concludes the paper.

II. An Overview of the Selected Open-Source GPGPU Projects

In order to understand how effectively application developers have been able to use the GPU hardware, we selected ten open-source GPGPU projects in our study. We listed these projects in Table I and also reported both the performance gains and energy savings after we fixed the performance bugs.

Many of these abovementioned projects are designed as libraries to be used as middleware. Therefore, additional performance enhancements and the energy savings have significant benefits to end users. These projects are selected given the criterion that they already have reasonable high performance based on either the published results or our own tests.

III. BACKGROUND ON GPU COMPUTING

Modern GPUs share similar many-core architecture. On-chip processor cores in GPUs are organized in a hierarchy. In NVIDIA G80 and GF100 (a.k.a Fermi) architecture, a GPU has a number of streaming multiprocessors (SMs) (30 SMs in a GTX 285 and 15 in a GTX480) and each SM contains multiple streaming processors (SPs) (8 in a GTX285 and 32 in a GTX480). Similarly, in an AMD/ATI HD5870 GPU, there are 20 SIMD engines (i.e., SMs) and each SIMD has 16 stream cores (i.e., SPs). Each SP in G80/GF100 is a scalar processor while each stream core in HD5870 has a 5-way VLIW (very-long instruction word) pipeline. The GPU on-chip memory resource includes register files (64kB per SM in GTX 285, 128kB per SM in GTX480, 256kB per SIMD in HD5870), shared memory (16kB per SM in GTX285, 16kB or 48kB in GTX480 depending on the configuration) or local data share (32kB per SIMD in HD5870), and caches for different memory regions, including texture and constant memory. In GTX480, there is a 16kB or 48kB L1 cache per SM for global memory, depending on the shared memory configuration. In HD5870, there is a line buffer per SIMD, which can be viewed as a limited cache for global memory as it only provides reuse for the accesses in the same wavefront.

TABLE I. Open-Source GPGPU Projects

Project	Description	Speedups (Energy savings)
DecGPU [5]	An error correction algorithm implemented in NVIDIA CUDA and MPI, and runs on a GPU cluster.	1.19x(1.20x) on GTX480
FLAGON [10]	A library for programming NVIDIA CUDA from Fortran 9x. It provides multiple primitive functions and an interface to CUBLAS and CUFFT library.	1.67x(1.48x) on GTX480
GPUMLib [12]	A GPGPU code library for machine learning algorithms.	1.78x(2.09x) on GTX480
Ising GPU [26]	A project uses GPUs to accelerate Monte Carlo simulation of the 2D and 3D Ising models. Up to 35X speedups are achieved over the CPU implementation.	1.52x(1.46x) on GTX480
MUMmerGPU [9]	A high-throughput parallel pair-wise local sequence alignment program; 13X faster than the CPU version.	1.05x(1.10x) on GTX 480
nDust [13]	A set of GPGPU programs to calculate dust-plasma charge equilibrium of dust-plasma systems in protoplanetary disc environments.	4.77x(4.91x) on GTX 480
OpenCurrent [19]	A C++ library for solving Partial Differential Equations (PDEs) over regular grids.	1.07x(1.08x) on GTX 480
Qymsym [21]	A GPU accelerated parallel hybrid symplectic integrator for planetary system integration.	1.43x(1.58x) on GTX 480
ViennaCL[29]	An OpenCL code library of common linear algebra operations and the solution of large sparse systems of equations by means of iterative methods.	1.71x(1.67x) on HD 5870 3.21(3.01) on GTX 480
CUBLAS & CUDA SDK	Although CUBLAS 3.1 [15] and 3.2 [16] are not open source, their matrix multiplication implementations [27] [23] are available. The matrix multiplication in CUDA SDK is [17] open source.	On GTX 480 CUBLAS 1.3x(1.13x), SDK1.26x(1.26x)

To hide long off-chip memory access latencies, a high number of threads are able to run concurrently on GPUs. These threads follow the single-program multiple-data (SPMD) model. The threads are grouped in 32-thread warps in NVIDIA GPUs with each warp being executed in the single-instruction multiple-data (SIMD) manner. In AMD/ATI GPUs, the equivalent term is a wavefront, which has 64 threads sharing a program counter (PC).

In the GPGPU programming model such as OpenCL and CUDA, the code to be executed by GPUs is kernel functions.

All the threads will run the same kernel with different thread/block identifiers (ids) to determine their workload. CUDA also defines a thread block (called a work group in OpenCL) as an aggregation of threads which must be executed in the same SM/SIMD engine and the threads in the same thread block can communicate with each other through the shared memory/local data share on the SM/SIMD engine.

Besides extracting data-level parallelism (DLP) from application algorithms, there are several important aspects to achieve high performance GPU computing. Here, we highlight those related to the performance bugs that we found in our study. The first is to access the global memory efficiently and the key is memory coalescing. The second is effective usage of the on-chip software-managed storage, which includes the register file and shared memory. Such resource is critical to leverage data reuse and reduce the overall number of memory accesses. The register file has lower latency than the shared memory while the latter can be used to support communication among threads in the same thread block. The third is effective usage of the hardware caches, which include the global memory cache (if available), the constant cache, and the texture cache. The fourth is to achieve a good balance between instruction-level parallelism (ILP) and thread-level parallelism (TLP).

IV. METHODOLOGY

In our study, we perform our experiments on three different GPUs, NVIDIA GTX285, GTX480, and AMD HD5870 to reveal interesting performance impacts of the performance bugs on different GPU architectures.

CUDA SDK 3.1 [17] is used in all the experiments on NVIDIA GPUs. In experiments on ATI/AMD Radeon HD 5870 GPUs, ATI Stream SDK v2.2 [2] is used. The host machines have 4GB memory and an Intel Core 2 Quad Q9650 CPU. To evaluate the power/energy consumption, we use a current probe [1] to detect the current and a power meter [22] to detect the voltage of the inputs of the whole system. The difference between the idle system and the active system running a program is used to determine the power/energy consumption of the program. We keep running one kernel multiple times for 30 seconds and collect the overall energy consumption and how many times that the kernel is executed. Then, we calculate the energy efficiency in the unit of GBytes per Joule (overall data transmitted in 30s / energy consumption of the kernel in 30s) for data transmission benchmarks or Gflops (overall floating point operations in 30s / energy consumption of the kernel in 30s) per Joule for computational workloads.

V. CHARACTERIZING AND FIXING PERFORMANCE BUGS

The wide range of GPGPU programs under our study ensures us to focus on algorithm-independent performance bugs. We group them into six categories. For each category, we will present the buggy code, the prognosis, our proposed fixes, and the results. In our discussion, we use either the extracted code from the projects/programs, which contain the bugs, or sample code segments to illustrate the problem and discuss our proposed fixes. The sample code uses either CUDA or OpenCL to indicate that our discussions apply to both programming models.

A. Global Memory Data Types and Access Patterns

The bugs in this category are mainly about memory coalescing. Since this issue has been addressed in recent works, we refer to prior works wherever possible and only elaborate the cases that have not been addressed before.

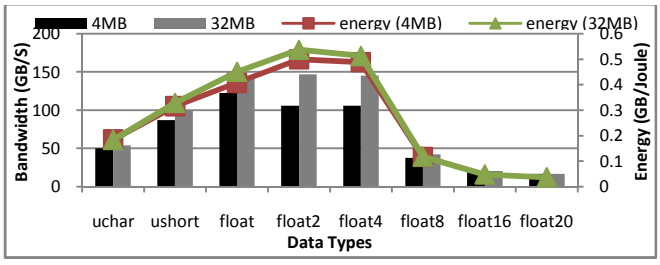

Figure 1. Bandwidth and energy efficiency for data types on GTX480.

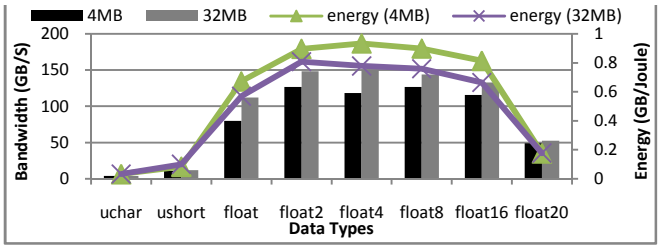

Figure 2. Bandwidth and energy efficiency for data types on HD5870.

1) Global Memory Data Types

Buggy Code This type of bugs is due to the use of data types that are either smaller than 'float' or larger than 'float4'. Here, we use 'floatn' to represent a structure data with a size of n floats.

Prognosis The problem with the data types either smaller than 'float' or larger than 'float4' is that they violate the memory coalescing requirements. We quantify the performance impact with a copy kernel to move a 1-dimensional (1D) array. We vary the data type and examine the sustained bandwidth for copying different amount of data. The results are shown in Figure 1 for GTX480 and Figure 2 for HD5870. GTX480 is configured that it has 48kB L1 cache and 16kB shared memory in each SM so as to maximize the cache effect. GTX285 has similar but lower performance compared to GTX480 due to its lower memory access bandwidth and the lack of the L1 cache. From Figure 1, we can see that GTX480 delivers much lower bandwidth when the data type is not 'float', 'float2', or 'float4'. In comparison, HD5870 achieves good bandwidth for data types as large as float16 (a vector/struct with 16 float numbers) as shown in Figure 2. The reason is that although the memory accesses are not coalesced, the large line size of the line buffers on HD5870 provides effective spatial reuse so that each fetched line is fully utilized. Therefore, no bandwidth is wasted until the data type is larger than 'float16'. On the other hand, for HD5870, the highest bandwidth is achieved when the data type is 'float2' while GTX480 has similar performance for the 'float', float2' and

'float4' data types. The energy efficiency of such data movement correlates well with the bandwidth results for both GPUs, as shown in Figures 1 and 2.

Fixes In order to support generic data types efficiently, the following two fixes are to be applied. First, for large data types (larger than 'float4' on GTX285 and GTX480 or larger than 'float16' on HD58470), we will use shared memory to convert the code so that the accesses can be coalesced, similar to the approach used in [31]. The key to the fix is to convert the desired data type to 'float' before accessing the data from the global memory and then to use a loop to the data into the shared in a coalesced manner. After the data is present in the shared memory, we can directly access the data using the desired data type. Since HD5870 has the best bandwidth for 'float2', we use 'float2' to load the data into the shared memory. Second, for global memory accesses using a data type smaller than 'float', the fix is to combine the workload of multiple threads so that we can use the 'float' type to access the global memory and then extract the desired datum for computation.

Results The performance of the copy kernel on all GPUs is close to the peak bandwidth after we apply the proposed fixes to data types. The energy efficiency also correlates with performance.

```
float a = input[gid*2+0];
float b = input[gid*2+1];
output[2*gid+0] = a;
output[2*gid+1] = b;
```

(a) A code example with the adjacent data access pattern.

```
float sum = 0;
for (int i=0; i<width; i++){
sum += input[gid*width+i];
}
output[gid] = sum;
```

(b) A code example with the row-based data access pattern

```
int delta = 1;
float a_0 = input[gid+delta+16];//the offset 16 is to eliminate
float a_1 = input[gid+16];     // boundary checks
float a_2 = input[gid-delta+16];
```

(c) A code example with shared data access pattern

Figure 3. Kernel code examples with various data access patterns. The variable 'gid' is the global thread id.

2) Global Memory Data Access Patterns

Buggy Code Among the applications under our study, we found a few common data access patterns that may cause performance degradation. These patterns include adjacent accesses, row-based accesses, and shared data accesses. The buggy code examples are shown in Figure 3.

For adjacent data accesses, each thread accesses a few data items which are adjacent to each other, as shown in Figure 3a.The function 'get_global_id(0)' in OpenCL returns the global thread id (gid), equivalent to '(blockIdx.x*blockDimx+ threadIdx.x)' in CUDA.

In the row-based access pattern, each thread accesses a row of data sequentially, as shown in Figure 3b. It can be viewed as a more generic case of the adjacent access pattern by extending the range of adjacency.

In the shared access pattern, the threads in the same/different warp(s) may access the same data using different instructions. One example is shown in Figure 3c.

Prognosis The problem with the code in Figure 3a lies in the array indices. Taking the load access 'input[gid*2+0]' as an example, the threads in a half warp may access the following data: input[0], input[2], …, input[32], which fails the requirements for memory coalescing. Among the GPUs, GTX285 suffers the most. GTX480 alleviates the problem by keeping the data accessed by 'input[gid*2+0]' in the cache. When the next load instruction 'input[gid*2+1]' from the same warp executes, it hits in the cache. However, the cache is not effective for the indices [gid*4+0], [gid*4+1], etc. The reason is that the data accessed by the first load 'input[gid*4+0]' of a warp (32 threads) span a large range of data from input[0] to input[124]. When the load instruction 'input[gid*4+1]' from the same warp executes, some data have already been replaced, resulting in cache misses and overall low bandwidth throughput. Similarly, for HD5870, its line buffer cannot hold the data to handle such accesses.

Similar to the adjacent access pattern, the global memory access 'input[gid*width+i]' in Figure 3b fails to meet the memory coalescing requirement. Furthermore, with the variable 'width' larger than 4, the L1 cache in GTX480 offers no relief.

The problem with the code in Figure 3c is that it violates the alignment (e.g., input[17] ~ input[49]) required for memory coalescing and the same data are loaded multiple times by different instructions in the same/different warps. Interestingly, the hardware cache, either the L1 global data cache in GTX480 or the line buffer in HD5870, overcomes this limitation effectively. GTX285, on the other hand, suffers from the problem, and has 86.6 GB/s bandwidth when accessing 16 MB data.

Fixes We fix the performance problem due to adjacent data accesses by replacing them ('input[2*gid+0]' and 'input[2*gid+1]') with vector-based accesses (a single 'float2'). Similar code conversion to the 'float4' type fixes the problem with the access pattern 'input[gid*4+0], input[gid*4+1], etc'. Note that after conversion to vector access, we load all four or two float numbers depending on the vector size. Even in the case when the kernel does not use all the elements of the vector, vector-based accesses are still faster than the original non-coalesced accesses.

We resort to shared memory to convert the row-based accesses to meet the coalescing requirement, similar to the fix to accessing large data types.

Since the hardware cache makes GTX480 and HD5870 immune to the problem associated with the shared data accesses, the fix only applies to GTX285. The fix is also to use shared memory to achieve coalesced memory accesses, as discussed before. However, our measurements show the shared memory is more energy efficient (0.586 GB /J) than to the L1 cache (0.568 GB/J) on GTX 480, although they have the similar bandwidth. The difference is due to tag matches required by the L1 cache.

Results We examine the effect of bug fixes to Figure 3 by accessing 64MB data. Our results show that the fixes (using vectors) significantly improve the performance, up to

3.33X for GTX285, 1.56X for GTX480 and 2.66X for HD5870. Similarly using shared memory to fix the row-based data access bug, we can achieve performance improvement of 5.96X for GTX285, 3.46X for GTX480, and 36.9X for HD5870. The fix to shared data accesses for GTX285 can improve the performance by up to 2.44X.

```
int main() {
   dim3blkDim(16, 16); // Kernel invocation
   dim3gridDim(N / blkDim.x, N / blkDim.y);
   myKernel<<<gridDim, blkDim>>>(…);  }
```

Figure 4. Kernel invocation with a thread block dimension of 16x16.

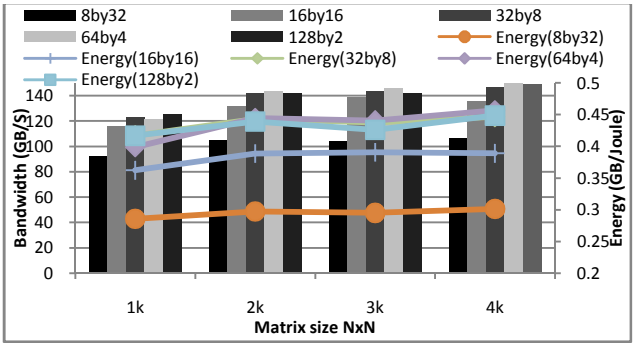

Figure 5. The bandwidth and energy efficiency of accessing 2D data (size NxN) using different thread block dimensions on GTX480.

B. Block Dimensions in Thread Hierarchy

Buggy Code In modern GPUs, threads are organized in a thread hierarchy, multiple threads forming a 1D/2D/3D thread block, and thread blocks forming a thread grid. Within the thread hierarchy, thread block dimensions, determining how threads are grouped in a thread block, are most important as the threads in the same thread block will be executed by the same SM and can communicate through the shared memory or hardware cache. As stated in the CUDA programming guide 3.1 [14], the block dimension of 16x16 is an arbitrary and common choice for applications with a 2D output domain. Therefore, it is no wonder that we encounter the buggy code shown in Figure 4 in the open-source projects under study.

Prognosis The problem with the code in Figure 4 is that the fixed block dimension like 16x16 does not consider how the data are accessed and reused and therefore is not the optimal choice. We examine three different data access and reuse patterns. The first is no data reuse among threads. The second is reuse through (software-managed) shared memory and the third is reuse through hardware-managed caches. Among them, the effect of thread block dimensions on shared memory usage has been observed before [11][24][27][30] and the other two, to our knowledge, have not been reported in the literature.

1) No data reuse

We again use a simple copy kernel to illustrate the performance effect of block dimensions on different GPUs. This time, the kernel moves a 2D array, which requires additional index computation than accessing a 1D array, resulting in lower bandwidths than reported in Figures 1 and 2. We keep the thread block size as 256 and only vary the dimension parameters. Increasing the thread block size has no impact while decreasing it to 128 will lead to resource under-utilization for GTX480. The reason is that each SM supports up to 1536 threads and up to 8 thread blocks. If each thread block has 128 threads, one SM cannot have more than 1024 (8x128) threads, wasting its resource.

Figure 5 reports the achieved bandwidth and the energy efficiency using different thread block dimensions on GTX480. GTX285 is less sensitive to the block dimension as long as the X dimension size (i.e., blockDim.x) is at least 16. From Figure 5, we can see that the block dimension of 16x16 fails to achieve the optimal performance. By simply changing the block dimension to 32x8, we have consistently better performance (up to 8%) and energy efficiency (15%). In contrast, if the thread block dimension is changed to 8x32, a half warp will access two different rows, violating coalescing requirements and resulting in poor performance and low energy efficiency. HD5870 shows the similar behavior and it prefers a larger block dimension along the X direction as the wavefront size is 64. Therefore, we can conclude that if there is no data reuse and the thread ids are used as global memory access addresses, the block dimension should be such that the X dimension is at least 32 for GTX480 and 64 for HD5870.

```
__global__ void matrixMul( output* C, input* A, input* B, intwA,
in wB){
    int tx = threadIdx.x; int ty = threadIdx.y;
    ….//variable declaration and definition
    for (a = aBegin, b = bBegin;a <= aEnd;a+=aStep, b+=bStep){
        __shared__ float As[blocky][blockx], Bs[blockx][blockx];
        …//load a tile of A into shared mem As
        …//load a tile of B into shared mem Bs
        for(i= 0; i<Step; i++)
            Csub += As[ty][i]* Bs[i][tx];
    }
    …//store Csub to matrix C
}
```

Figure 6. The tiled matrix multiplication based on the sample code in NVIDIA CUDA SDK.

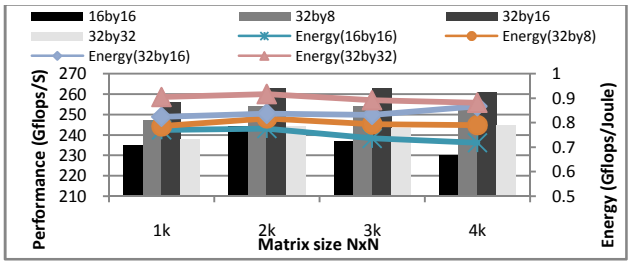

Figure 7. The throughput and energy efficiency of tiled matrix multiplication for different thread block dimensions on GTX480.

2) Data reuse through shared memory

When the data are explicitly reused through shared memory, both global memory access bandwidth and data reuse patterns need to be considered. We use the matrix multiplication code provided in the CUDA SDK as our case study. The kernel is a tiled version of matrix multiplication (C = A x B) with an equal tile size along the X and Y

direction. The block dimension is set up as 16x16. To make the code run with different thread block dimensions, we slightly change the kernel so that it differentiates the tile size along the X and Y directions, as shown in Figure 6. Then, we try with the following block dimensions on GTX480, 32x8, 32x16, and 32x32, and compare to the block dimension of 16x16. The GTX480 is configured to have 48kB shared memory for each SM as the kernel explicitly uses the shared memory for data reuse. The results are presented in Figure 7.

From Figure 7, we can see that if we keep thread block size as 256 and change the block dimension from 16x16 to 32x8, the same kernel code can achieve higher performance, ranging from 5.1% for the matrix size of 1kx1k to 10.0% for the matrix size of 4kx4k. The reasons are (1) the 32x8 block dimension has better global memory access bandwidth, as shown in Figure 5; (2) the data reuse pattern has been changed. The code with the 32x8 block dimension essentially implements a tiled version with uneven tile sizes. The tile size of matrix A is 32x8 and the tile size of matrix B is 32x32. From Figure 6, we can see that after the tiles from both A and B are loaded into the shared memory, they will be used by all the threads in the thread block. Based on the access, As[ty, k], we can see that each element in the tile from matrix A is reused by the threads with the same tx (i.e., thireadIdx.x). Similarly, each element in the tile from matrix B is reused by the threads with the same ty, as a result of the access Bs[k, tx]. For block dimension of 16x16, each element in either tile is reused 16 times. However, when the block dimension is changed to 32x8, the tile from matrix A is of size 32x8 (As[8][32]) and is reused 32 times while the tile from matrix B is of size 32x32 (Bs[32][32]) and is used 8 times. Based on such reuse patterns, we can see that when we change the thread block dimension from 32x8 to 32x16, we can further improve the reuse of the tile from matrix B, thereby achieving higher performance. On the other hand, when the block dimension is increased to 32x32, although data reuse is improved, each thread block will have 32x32=1024 threads. As a result, one SM can only support one such thread block, leading to resource underutilization. Therefore, the best performance is achieved with the block dimension of 32x16, which shows consistently higher performance (12% on average) than the block dimension of 16x16 used in CUDA SDK. Energy efficiency, on the other hand, does not exactly follow the trend of performance. Instead, it is a combination of performance and the number of off-chip memory accesses for this application. With the block dimension of 32x32, the high data reuse leads to the best energy efficiency although the performance is suboptimal. The block dimension of 16x16, in comparison, has the lowest energy efficiency.

3) Data reuse through hardware cache

For applications with complicated access or reuse patterns, it is challenging to explicitly manage shared memory for data reuse. In this case, hardware caches provide a user friendly way to exploit temporal and spatial locality for data reuse. We show that block dimensions have a significant impact on how effective the cache is utilized. We use the activation matrix kernel from the GPUMLib [12], shown in Figure 8, as an example to examine the interaction between the block dimensions and cache utilization. The block dimension of 16x16 is used in the open-source library to invoke this kernel.

```
__global__ void ActivationMatrix(output *Output, input *A, input
*B, intwA, intwB){ …
   int idnx = blockIdx.x*blockDim.x + threadIdx.x;
   int idny = blockIdx.y*blockDim.y + threadIdx.y;
   for(i=0; i<wA;,i++){
      a = A[idnx * wA + i];
      b = B[idny * wB + i];
      temp  +=  pow(a - b , 2);
   }      …}
```

Figure 8. The activation matrix kernel from GPUMLib [12].

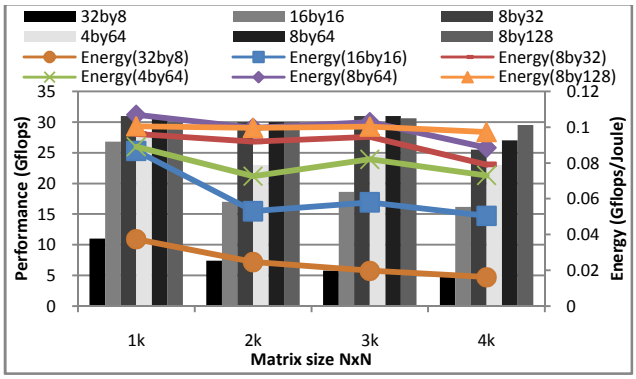

Figure 9. The performance of the activation matrix kernel (wA=wB=N) using different thread block dimensions on GTX480.

For the kernel shown in Figure 8, we make no modification and only vary the block dimensions. The GTX480 GPU is configured to have 48kB L1 cache for each SM as the kernel does not use any shared memory and relies on the cache for data reuse. We first keep the same thread block size (256) in a thread block and vary the dimensions from 4x64 to 32x8. Then, we increase the thread block size and use the dimensions of 8x64. The results are shown in Figure 9.

From Figure 9, it is apparent that the thread block dimension has significant performance impacts. Among the block dimensions with 256 threads in a thread block, the dimension 8x32 is the best and it consistently outperforms the original dimension of 16x16 (up to 77%) and has higher energy efficiency (up to 87%). To understand how the performance gains are achieved, however, is non-trivial. The reason is that although the code shown in Figure 8 is relatively simple, its data reuse pattern and the interaction with GPU hardware are complicated. For the two global memory accesses, A[idnx * wA + i] and B[idny * wB + i], in the kernel, there are various types of data reuses. When the thread block dimension is 8x32, the thread id along the X direction, threadIdx.x, ranges from 0 to 7, and the thread id along the Y direction, threadIdx.y, ranges from 0 to 31. When we consider reuse within a thread block, the access B[idny * wB + i] will have reuses among the threads with the same threadIdx.y and different threadIdx.x. Therefore, it will be reused 8 times when the dimension is 8x32. Moreover such reuses are intra-warp reuses as these threads

with the same threadIdx.y and different threadIdx.x are in the same warp. In comparison, the reuses for the access A[idnx * wA + i], are among the threads with the same threadIdx.x and different threadIdx.y. Therefore, for the block dimension of 8x32, the access can be used 32 times and the reuses include both intra-warp and inter-warp reuses since these threads have different threadIdx.y. The implication of intra-warp reuses is that the cache block can be replaced right after it is accessed since all the threads in a warp are executed in the SIMD mode. To exploit inter-warp reuses, the cache block cannot be replaced until the other warps access it. Therefore, inter-warp reuses require more cache capacity than intra-warp reuses. Overall, for the block dimension of 8x32, each thread block requires 8 cache blocks to provide reuses of A[idnx * wA + i], since there are 8 different threadIdx.x, and 4 cache blocks to provide reuses of B[idny * wB + i], since there are 4 different threadIdx.y in each warp and those reuses are intra warp reuses. For the block dimension of 16x16, we perform the similar analysis and reveal that a thread block requires 16 cache blocks for reusing A[idnx * wA + i] and 2 cache blocks for reusing B[idny * wB + i]. The lesser requirement for cache capacity of the thread block dimension 8x32 helps to explain why it achieves higher performance. The actual caching effects are more complicated as either access will go through a row of length 'wA', which can be a large number, so that much more data need to be cached. Furthermore, if we consider reuses among threads across different thread blocks, the access B[idny * wB + i] may also have inter-block reuses if two thread blocks with the same blockIdx.x are scheduled on the same SM as the cache is shared among the thread blocks scheduled on it.

After determining that the thread block dimension of 8x32 provides the best performance among those with the same thread block size, we increase the thread block dimension along the Y direction to increase the reuses of A[idnx * wA + i]. The results show that for large matrices, e.g., 4kx4k, the block dimension 8x64 achieves better performance than 8x32. Note that although the block dimension of 8x64 has similar performance to 8x32 for small matrices, the improved data reuse leads to higher energy efficiency. When the dimension is further increased to 8x128, each SM can only run one thread block, resulting in lower performance.

Fixes Based on the discussion above, we propose the following fixes: (1) to maximize the global memory access bandwidth, if the thread ids are used as memory access addresses, the size of the X dimension in a thread block needs to be at least 16 for GTX 285, 32 for GTX480, and 64 for HD5870; (2) to improve data reuses, either using software managed shared memory or hardware managed caches, the thread block dimensions need to be explored and the optimal configuration can be application and data size dependent; (3) The improved data reuse leads to lower power consumption but not necessarily performance. The overall energy efficiency depends on both performance and power consumption.

Results As presented in Figures 5, 7, and 9, we can significantly improve the application performance by simply varying the block dimensions. Compared to the commonly used block dimension of 16x16, the performance of the tiled matrix multiplication and activation matrix kernel can be improved by up to 13% and 77%, respectively, and the energy efficiency can be improved by up to 22% and 87%, respectively.

```
__kernel void tmv(__global float *A, __global float *B, ……){
  __local float shared_1[blockDimX];
  uint y = get_global_id(0);
  floatdotProduct = 0;
  for (uint x = 0; x < height; x = x + blockDimX){
     shared_1[(tidx+0)]=B[(x+tidx)];
     barrier(CLK_LOCAL_MEM_FENCE);
     for(uint i = 0; i < blockDimX; i ++)
       dotProduct += A[y + (x+i)*width] * shared_1[i];
     barrier(CLK_LOCAL_MEM_FENCE);
  }
  C[y] = dotProduct; }
```

Figure 10. An optimized (but buggy) kernel of transpose-matrix-vector multiplication (tmv).

C. Code Portability

Buggy Code For some projects under our study, one objective is to provide portability across both AMD and NVIDIA GPUs. However, it is not recognized that different hardware features in different GPUs necessitate significantly different performance considerations. One example is the OpenCL implementation for transpose-matrix-multiplication (tmv) C=AT*B, where A is an M×N matrix and B is an M-dimension vector, as shown in Figure 10. The code in Figure 10 is already well optimized with the tiling technique. The shared memory is also used to exploit the reuse of the vector B and all the global memory accesses are coalesced.

```
__kernel void tmv(__global float *A, __global float *B, ……){
  __local float shared_1[blockDimY][blockDimX];
  __local float shared_2[blockDimY][blockDimX];
  int tidx = get_local_id(0); int tidy = get_local_id(1);
  float t; float sum = 0;
  for(uinti = 0; i< height; i = i + blockDimY*blockDimX){
     shared_1[tidy][tidx] = B[i+tidy*blockDimX+tidx];
     barrier(CLK_LOCAL_MEM_FENCE);
     for(uint j =0; j <blockDimX; j++) {
        t = A[y+(tidy*blockDimX+j+i)*width];
        dotProduct += t*shared_1[tidy][j];
     }
     shared_2[tidy] [tidx] = dotProduct;
     barrier(CLK_LOCAL_MEM_FENCE);
     if(tidy == 0){ //reduction
        for(uint i = 1; i < blockDimY; i++)
           dotProduct += shared_2[i] [tidx];
        C[y] =  dotProduct;}
   }  }
```

Figure 11. A fix to limited TLP by partitioning the workload of each thread in Figure 10 into multiple threads.

Prognosis There are two main issues with the code shown in Figure 10. First, the 'float' data type is used. As shown in Figures 1 and 2, the 'float' data type is well supported on GTX480 but HD5870 prefers the 'float2' type. Second, different GPUs favor different ILP vs. TLP tradeoffs. Between GTX480/285 and HD5870, GTX480/285 prefers a high number of light-weighted threads while HD5870 is

more efficient when each thread has enough computations to populate its 5-way VLIW pipeline. For the kernel code in Figure 10, each thread computes a dot product of one column in the matrix A and the vector B and generates one element in the product vector C. For the matrix size of 1k×1k, it means that there are 1k threads and each thread computes the dot product of two 1k-entry vectors. For both GTX480/285 and HD5870, the number of threads is too low to hide the memory latency. As we can see from Figures 12, even when the matrix width increases to 4k, the number of threads is still insufficient.

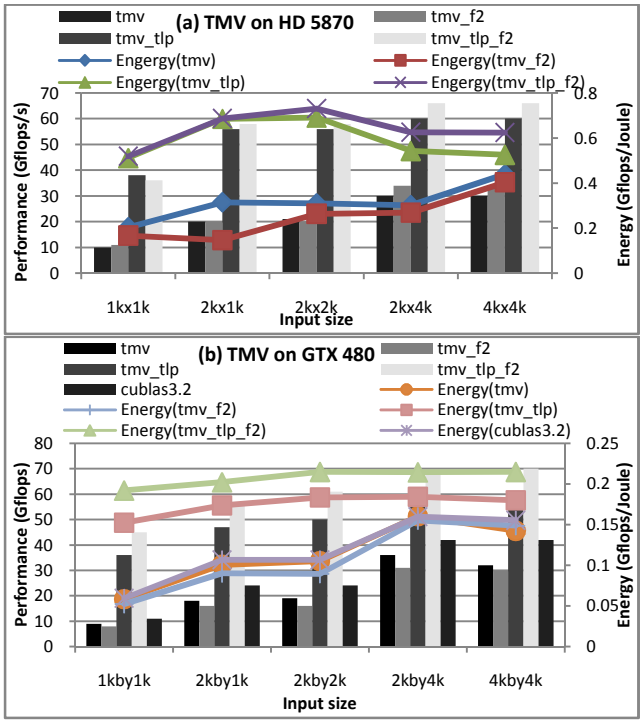

Figure 12. The throughput and energy efficiency of tmv on (a) HD5870 and (b) GTX480. The label '_f2' means the results using the 'float2' type; '_tlp' means the results with the fix shown in Figure 11; '_tlp_f2' means the results with both fixes.

Fixes Given the two problems with code in Figure 10, we propose two fixes. The first is to use the 'float2' type to access the global memory as it is well supported in both GTX285/480 and HD5870. The use of float2 also enforces that each thread compute two adjacent elements in the product vector. Although this fix reduces the number of threads (i.e., reducing TLP) and increases the workload of each thread (i.e., increasing ILP), there also may exist additional benefit. In tmv, since each thread now computes two dot products, each element in the vector B is used twice, thereby reducing the overall number of shared memory accesses (B is accessed from shared memory). The second fix is to generate more threads to address the limited TLP problem. In tmv, instead of letting one thread compute the dot product of two long vectors, we break the workload into multiple threads with each thread computing part of the dot product. Then, a reduction operation is performed to combine the partial dot-product results, similar to the approach used in [11]. The fix is shown in Figure 11. These two fixes can usually be combined so that applications can benefit from high bandwidth provided by vector data types and also compensate the lost TLP due to vectorization.

Results The results of our proposed fixes to the code in Figure 11 are shown Figure 12 for HD5870 and GTX480. Following the proposed fixes in Section V.B, the group dimension (i.e., thread block dimension) is set as 64x4 for HD5870 and 32x8 for GTX480. From the figures, we can see that the first fix alone is not very effective for small matrices due to the reduced TLP. The second fix is always beneficial. When the two fixes are combined, the increased TLP from the second fix makes up the TLP loss from the first and the best performance is achieved. Overall, with our proposed fixes, the performance of the transpose-matrix-vector multiplication is improved by 2.5X to 3.8X on HD5870, 2.0X to 5.0X on GTX480. As a reference, our code achieves 1.7X to 4.0X higher performance on GTX480 than CUBLAS 3.2. The results on GTX285 are similar to GTX480. The trend of energy efficiency is similar to the performance trend.

D. Constant and Texture Memory

Buggy Code Among the projects under our study, only few use constant or texture memory. In these few projects, although the use of constant or texture memory is functionally correct by supplying read-only data, we are not aware that any exploits the performance benefits of them, constant memory in particular.

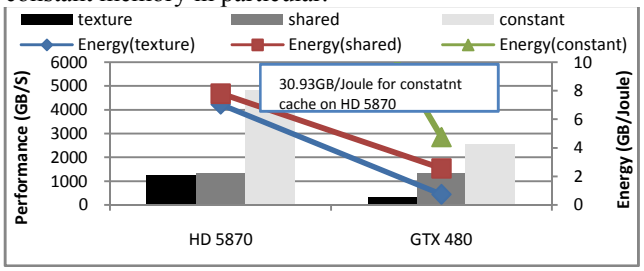

Figure 13. Bandwidth and energy efficiency comparison among various caches in GTX480 and HD5870.

Prognosis Proper use of constant and/or texture memory can render extremely high bandwidth due to the on-chip constant and texture caches. We developed a micro benchmark to characterize the performance of these caches and compare them to shared memory. In the micro benchmark, all the threads access the same small amount of data over and over again, thereby hitting in the caches repeatedly. The results are presented in Figure 13 for GTX480 and HD5870. From the figure, we can see that in both HD5870 and GTX480, the constant cache has much higher bandwidth than shared memory (4.83X and 2.05X, respectively). The texture cache has higher bandwidth than shared memory in HD5870 while lower bandwidth in GTX480. GTX285 has similar but lower bandwidth results than GTX480.

Fixes Given the high bandwidth of the constant cache in both GTX480 and HD5870, we propose to access large amount of data through constant memory rather than the common usage of supplying a small set of global constant

parameters. The key is to access the data allocated in constant memory one batch a time so that the data to be accessed can fit in the constant cache. We propose a novel implementation of matrix multiplication using constant memory, which can outperform CUBLAS 3.1 and 3.2.

```
__global__ void kernel_mm(float * B, float *C,…) {
  int idx = blockIdx.x*blockDim.x + threadIdx.x; float sum[16];
  for (int i=0; i<128; i++) {
      float b = B[i] [idx];
      for (int j=0; j<16; j++)
          sum[j] += b*CONSTA_A[i*16+j];//A in constant mem.
  }
  for (int j=0; j<16; j++)  C[j] [idx] = sum[j]; }
```

Figure 14. The kernel function for matrix multiplication using constant memory.

```
for (int j=0; j<WidthA; j+=WidthConst) {
  for (inti=0; i<HeightA; i+=HeightConst){//go through the tiles
  cudaMemcpyToSymbolAsync(const_0, …,
        cudaMemcpyDeviceToDevice,  stream[0]);
   //both const_0 and const_1 are const buffers
   kernel_0<<<grid, threads, 0, stream[0]>>>(B+offset_B0,
        C+offset_C0);
   cudaMemcpyToSymbolAsync(const_1, …,
        cudaMemcpyDeviceToDevice,  stream[1]);
   kernel_1<<<grid, threads, 0, stream[1]>>>(B+offset_B1,
      C+offset_C1) ; ……
}}
```

Figure 15. The code to invoke two concurrent kernels and both kernels run the same code as in Figure 14.

The main idea of our approach is to adapt the classical tiling algorithm. For matrix multiplication C=A×B, rather than using the same tile size for both A and B, we break down A into small tiles so that each tile in A can fit in the constant cache. The matrix B remains in the global memory and its tile size is much larger. For each element loaded from the matrix B (e.g., B[0][0]), we want to reuse it as much as possible. For the element B[0][0], it will be used to compute A[0][0]*B[0][0], A[1][0]*B[0][0], A[2][0]*B[0][0], etc., Although a tile of A can fit in the constant cache, such column major accesses are not as efficient as row-major accesses. Therefore, after we extract a tile from A, we perform a transpose operation before we copy it to the constant memory. Then, we perform the matrix multiplication on the tile that is present in the constant memory with the kernel code shown in Figure 14. The code in Figure 14 is based on the tile size of A being 16x128. The tile size of B is 128×widthOfB. Each thread computes one column (i.e., 16 elements) of the product of the two tiles. From Figure 14, we can also see that the accesses to the constant 'CONST_A[i*16+j]' is independent on thread id, thereby taking full advantage of the broadcast logic to broadcast the value to all the threads in the same warp.

To fully utilize the GPU resource, we also leverage the independence among different tiles in the product matrix C by invoking multiple kernels concurrently through multiple streams/command queues, which are an interesting feature in CUDA/OpenCL to support concurrent kernels. The code segment is shown in Figure 15, in which both kernels have the same code as shown in Figure 14 while the kernel 'kernel_0' is associated with the constant buffer 'const_0' and the kernel 'kernel_1' is associated with the constant buffer 'const_1'. The asynchronous memory copy from global to constant memory also helps to overlap the memory copy latency with actual computation. Our detailed implementation is released as open source code as well.

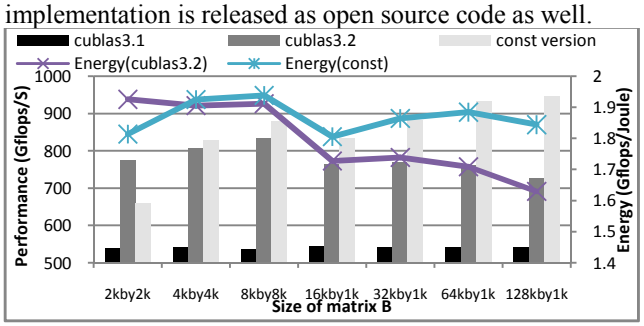

Figure 16. The performance of matrix multiplication C=AxB. We vary the width of input matrix B.

Results In Figure 16, we compare the performance of our proposed implementation (the time for transposing small tiles is included) to CUBLAS 3.1 and CUBLAS 3.2 on GTX480. From the figure, we can see that our proposed scheme outperforms CUBLAS 3.1 by up to 74% and reaches 945 GFLOPS. Compared to CUBLAS 3.2, our approach also achieves up to 30% speedup when the matrix size is 128kx1k. Therefore, our approach is particularly beneficial to scientific computation on large matrices. For example, the input matrix is 2240000 by 2240000 for Linpack [4]. As a reference, the matrix multiplication from AMD Stream Computing SDK uses texture memory and achieves 1.6 TFLOPS for 4kx4k matrices due to its larger register file and higher peak computational throughput (2.72 TFLOPS vs. 1.35TFLOPS) than GTX480. As GTX285 does not support concurrent kernel execution, our approach cannot be applied on GTX285. We expect that our proposed scheme will also work well on HD5870 as its constant cache is faster than its texture cache. However, the current ATI SDK does not support concurrent kernel execution either.

E. Function Specialization

Buggy Code and Prognosis For some applications, certain parameters have a limited range of possible values. The problem with the buggy implementation is that such parameters are declared as variables, which limits the compiler's capability to optimize the code. Such a code sample is shown in Figure 17. The project documentation states that based on the application, the constant variable 'KmerSize' will be an odd number and is less than 33. The default value is 31.

Fixes and Results In order to improve the performance of the code in Figure 17, we leverage the limited value range of 'KmerSize' by declaring a template so that we can generate multiple kernels with each having a different fixed value of 'KmerSize'. Then, we replace the shared memory array with registers, as shown in Figure 18. With this fix, the performance of this kernel is improved by 1.93X on GTX480.

```
__global__ void globalGetErrorPos(){
    extern __shared__ uchar space[];
    //KmerSize is from constant mem
    for (i = 0; i<KmerSize; ++i) {
        uchar ch = (off != i) ? space[i*TH_PER_TB+tid] : sub; …
}…}
```

Figure 17. A code sample with an array in shared memory with its size determined by a constant variable.

```
template<int KmerSize>// we generate kernels for different sizes
__global__ void globalGetErrorPos(){
    uchar space[KmerSize];          // declare registers
    #pragma unroll    // unroll loop
    for (i = 0; i<KmerSize; ++i) {
        uchar ch = (off != i) ? space[i] : sub;   …
} …}
```

Figure 18. A fix to code in Figure 17 by replacing shared memory array with registers and enabling loop unrolling.

F. Floating-Point Number Computations

Buggy Code and Prognosis For applications involving computations on floating-point numbers, data accuracy and the order of computations can have significant performance impact. For the CUDA code "float t=…; float exp_dH_4 = exp(-(4.0) / t);", two issues exist. First, the CUDA compiler will treat the constant 4.0 as a double-precision number. Second, the computation (-(4.0) / t) may not be able to utilize the specialized hardware/software support for reciprocal.

On the AMD GPUs, we find that one double-precision multiply needs 4 MUL_64 instructions while double-precision add only needs 2 ADD_64 instructions. Furthermore, if a multiplication has one double precision float operand and an integer operand, it takes more than 10 ALU instructions to convert integer to double-precision float.

Fixes and Results Depending on the intention of application developers, there are two possible fixes. (1) If single-precision provides sufficient accuracy, we can use the explicit single-precision floating-point number (4.0f), which results in a 58% performance improvement. (2) If double precision is needed for the intermediate results (i.e., the parameter of the exponential function), we can replace (-(4.0) / t) with (-(1.0 / t) * 4.0) to explicitly take advantage of the specialized support for the reciprocal function. This simple fix introduces a 14% performance gain on GTX480. Due to the overhead of double-precision multiplication on AMD GPUs, loop strength reduction can be used to convert multiplications into adds to save ALU instructions and remove additional conversion from integers to double-precision floats. In the loop of MonteCarloAsianDP (a program in AMD APP SDK), we convert the code from a = b* i to a += b, where both a and b are double-precision floats and i is the loop iterator. Such a simple conversion reduces 15 ALU instructions, which is translated to 2% performance improvement.

G. Impacts of the Bugs and Fixes

In Table II, we summarize how often the bugs are present in the open-source projects under our study and also the performance gains when the fixes are applied. Some of the buggy code is in CUDA, therefore having no results on HD5870. For GTX285, its limited register file size and its lack of data caches, texture/constant memory as well as double-precision support are the reasons why some of our fixes cannot be applied. As shown in the table, among the 10 projects, we found 7 of them having bugs on global memory data types or access patterns. However, the fixes only achieve significant speedup on one project. The reason is that the global memory accesses are not the bottleneck for the remaining projects. Bank conflicts in shared memory have been also well studied and we do not find that it is a performance bottleneck in these projects. For example, Qymsym uses the data type 'double4' to access shared memory, which introduces bank conflicts. However, there is no performance benefit after such bank conflicts are removed as they are not the performance bottleneck.

TABLE II. A SUMMARY OF BUGS IDENTIFIED IN THE PROJECTS AND THE PERFORMANCE IMPACTS OF THE FIXES.

Bug type	Affected projects	Fixed kernels	Speedup GTX285	Speedup GTX480	Speedup HD5870
Global Mem.	7	1	11.14X	2.33X	31.30X
Thread block Dim.	10	4	N/A	1.07X-1.77X	N/A
Portability	1	1	1.82X-2.38X	1.61X-5.00X	3.80X-6.89X
Constant and texture	2	2	2.42X	1.1X-4.03X	9.30X
Function special.	3	3	N/A	1.93X-4.72X	N/A
Floating-point Num.	2	2	N/A	1.14X-1.50X	N/A

For thread block dimensions, all the projects use a fixed configuration of 16 by 16 (for 2D output domain), 256x1, 512x1, or 64x1 (for 1D output domain) without considering the different kernel characteristics and different hardware architectures. We improved four CUDA kernels by changing the dimensions on GTX480.

Among the projects under study, only two considered portability using the cross-platform OpenCL. Both of them, however, do not consider the hardware differences between the AMD and NVIDIA GPUs. After we fix them using the approaches discussed in Section V.C, we achieve significant performance improvements on all GPUs.

We use texture or constant memory for two projects. Our constant memory version of matrix multiplication outperforms CUBLAS 3.2 by up to 1.3X. Using texture memory improves the performance of matrix-vector multiplication in ViennaCL by 2.42X on GTX285, 4.03X on GTX480, and 9.30X on HD5870.

We improved three projects by applying function specialization. The CUDA kernel speedups are from 1.93X to 4.72X on GTX480. We identified two projects whose double-precision floating-point computations are accelerated by explicitly using the support for the reciprocal and reciprocal-square-root functions. The corresponding kernel speedups are 1.14X-1.5X on GTX480.

Next, we summarize the overall improvement in both performance and energy efficiency at the project level. Among the projects in Table I, five projects are libraries,

which provide many kernels to be used to the programmers, and we report our improvements based on the kernels that we fixed. For the remaining projects, as they run as a single application, we report the overall improvement. As shown in Table I, our proposed fixes achieve significant improvements on both performance and energy efficiency.

VI. Related work

To our knowledge, our work is the first to study a wide range of open-source GPGPU projects to categorize common performance issues. Among them, global memory access patterns have been well studied in [3][6][14][31][28] and we highlighted the impact of the recently introduced L1 data caches. For thread block dimensions, Liu et al., [30] proposed an empirical search to find optimal thread block dimensions for different input sizes. In our work, we analyze the block dimension impact based on whether there is data reuse and what hardware structure provides reuse and our proposed fixes (the proper block sizes along the X direction) significantly narrow down the search space of optimal thread block dimensions. Although a goal of OpenCL is to provide a cross-platform standard and [7][20][24] have studied some performance portability issues, we proposed a set of unique optimization strategies to achieve good performance on both AMD and NVIDIA GPUs. While Volkov et al. [27] showed that the data reuse in register and shared memory is the key to achieve high performance for matrix multiplication on GPUs, our proposed implementation leverages an overlooked resource, the constant memory, for reuse. In addition, we convert the data/thread-level parallelism into task-level parallelism by exploiting the latest support for concurrent kernel execution. Furthermore, we characterize the energy efficiency and show that our proposed fixes achieve both performance gains and energy savings.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we present an empirical study on ten open-source GPGPU projects so as to understand how well the developers have utilized the GPU resource. From our detailed inspection, we identified quite a few performance bugs. We characterize these bugs and propose corresponding fixes to significantly improve the performance and energy efficiency. Besides the direct impact on the GPGPU projects under our study, our fixes serve as a reference to prevent similar bugs in future GPGPU code library/tool development. Among the different types of bugs that we identified, most of them can be handled effectively by the compiler. Function specialization, in particular, involves identifying the most commonly used values for certain parameters and then optimizing the kernel using each of the values. Run-time profiling can be used to determine these values so that the compiler can optimize the code dynamically. This is part of our on-going future work.

Acknowledgment

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

References

[1] Agilent 1146AAC/DC Current Probe. Agilent.

[2] ATI Stream Software Development Kit v2.2. 2010.

[3] B. Jang, et al., Exploiting Memory Access Patterns to Improve Memory Performance in Data Parallel Architectures, IEEE TPDS, 2010.

[4] C. Yang, et al., Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing, IEEE Cluster, September, 2010

[5] Decgpu, http://sourceforge.net/projects/decgpu/

[6] I-Jui Sung, et al., Data Layout Transformation Exploiting Memory-Level Parallelism in Structured Grid Many-Core Applications, PACT, 2010.

[7] K. Komatsu, et al., Evaluating performance and portability of OpenCL programs. iWAPT, 2010.

[8] M. Alexander and C. Q. Alice. QYMSYM: A GPU-Accelerated Hybrid Symplectic Integrator That Permits Close Encounters. arXiv:1007.3458v1, 2010.

[9] M. Schatz, et al., High-throughput sequence alignment using Graphics Processing Units. BMC Bioinformatics, 8(1):474, 2007.

[10] N. A. Gumerov, et al., Middleware for Programming NVIDIA GPUs from FORTRAN 9X. Poster at SC'2007.

[11] N. Fujimoto, Dense matrix-vector multiplication on the CUDA architecture, Parallel Processing Letters, 2008.

[12] N. Lopes, et al., GPUMLib: A New Library to Combine Machine Learning Algorithms with Graphics Processing Units, 10th Conf. on Hybrid Intelligent Systems, 2010.

[13] NDust, http://code.google.com/p/astro-attic/wiki/NDustReadMe

[14] NVIDIA CUDA C Programming Guide 3.1. 2010.

[15] NVIDIA CUDA CUBLAS Library 3.1.May 2010.

[16] NVIDIA CUDA CUBLAS Library 3.2.Oct. 2010.

[17] NVIDIA GPU Computing SDK 3.1, http://developer.nvidia.com/gpu-computing-sdk, 2011

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

[19] Opencurrent, http://code.google.com/p/opencurrent/

[20] P Thoman, et al., Automatic OpenCL device characterization: guiding optimized kernel design. EuroPar 2011.

[21] Qymsym, http://astro.pas.rochester.edu/~aquillen/qymsym/

[22] P3 P4460, http://www.p3international.com/products/p4460.html

[23] R. Nath, et al., An improved MAGMA GEMM for Fermi GPUs, University of Tennessee Computer Science Technical Report, UT-CS-10-655, July, 2010.

[24] S. Rul, et al., An experimental study on performance portability of OpenCL kernels. SAAHPC, 2010.

[25] S. Ryoo, et al., Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. PPoPP 2008.

[26] T. Preis, et al., GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. J. of Comp. Physics, 2009.

[27] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. SC, 2008.

[28] Victor W. Lee, et al., Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. ISCA 2010.

[29] Viennacl, http://viennacl.sourceforge.net/

[30] Y. Liu, et al., A Cross-Input Adaptive Framework for GPU Programs Optimization. IPDPS, 2009.

[31] Y. Yang, et al., A GPGPU Compiler for Memory Optimization and Parallelism Management, PLDI, 2010.