A Code Merging Optimization Technique for GPU

Ryan Taylor and Xiaoming Li

ECE Department, University of Delaware, USA

Abstract. A GPU usually delivers the highest performance when it is fully utilized, that is, programs running on it are taking full advantage of all the GPU resources. Two main types of resources on the GPU are the compute engine, i.e., the ALU units, and the data mover, i.e., the memory units. This means that an ideal program will keep both the ALU units and the memory units busy for the duration of the runtime. The vast majority of GPU applications, however, either utilize ALU units but leave memory units idle, which is called ALU bound, or use the memory units but idle ALUs, which is called memory bound, and rarely attempt to take full advantage of both at the same time.

In this paper, we propose a novel code transformation technique at a coarse grain level to increase GPU utilization for both NVIDIA and AMD GPUs. Our technique merges code from heuristically selected GPU kernels to increase performance by improving overall GPU utilization and lowering API overhead. We look at the resource usage of the kernels and make a decision to merge kernels based on several key metrics such as ALU packing percentage, ALU busy percentage, Fetch busy percentages, Write busy percentages and local memory busy percentages. In particular, this technique is applied at source level and does not interfere with or exclude kernel code or memory hierarchy optimizations, which can still be applied to the merged kernel. Notably, the proposed transformation is not an attempt to replace concurrent kernel execution, where different kernels can be context-switched from one to another but never really run on the same core at the same time. Instead, our transformation allows for merged kernels to mix and run the instructions from multiple kernels in a really concurrent way. We provide several examples of inter-process merging describing both the advantages and limitations. Our results show that substantial speedup can be gained by merging kernels across processes compared to running those processes sequentially. For AMD's Radeon 5870 we obtained an average speedup of 1.28 and a maximum speedup of 1.53 and for NVIDIA's GTX280 we obtained an average speedup of 1.17 with a maximum speedup of 1.37.

1 Introduction

Today's GPUs consist of hundreds of processing cores and can run thousands of threads at the same time. Even though the sheer scale of parallelism may already generate good speedups for GPU programs, crucial resources on GPUs such as ALU processing units and memory units are frequently imbalancedly utilized, which prevents the realization of full GPU performance. The reason is that even though many threads can run on the GPU concurrently, the threads are identical copies of the same code. If a batch of threads saturate the ALU units but leave the memory units idling, the imbalance and under-utilization won't be remedied by context switching because the next batch of threads will use the resources in the same way. This observation is true for both NVIDIA's and AMD's GPUs. Clearly, GPUs can deliver better performance if the GPU resources are used in a more balance way.

Recently NVIDIA released a technology called concurrent kernel execution [1] that allows different kernels (i.e., different code) to be kept active on NVIDIA Fermi GPU and switches the execution between kernels when one kernel is stalled. While sounds like a perfect solution for resource idling, the concurrent kernel execution in fact does *not* improve the resource imbalance or resource under-utilization within a kernel because at any moment, only *one* kernel can run on a core of a GPU. While the resource idling due to long latency operations is averted by context switching, the imbalance of the kernels, utilization of the core resources is untouched. For example, assuming that we have two kernels, one only using ALU and one only doing memory operation. The concurrent-kernel-execution will switch back-and-forth between the two kernels because the memory operations are slow and will stall execution. Overall, the effective memory access latency is reduced. However, at any given moment, the GPU is either idling the ALU or idling the memory controller, because only one kernel runs at a time. In other words, the switching from an imbalanced kernel to another imbalanced kernel will not help either one to get more balanced. This sub-kernel level of resource imbalance and the consequent resource under-utilization is our key observation and is not addressed in current technology. We need a technique that works within a thread/kernel to help achieve full utilization of each of the given resource units in the GPU.

In this paper, we propose a novel code transformation technique that strategically merges kernels with different resource usage patterns to improve performance of GPU programs. This technique is applied at a coarse grain level. More specifically, this paper makes three contributions. The first contribution is the identification of a group of profiling metrics that defines how *fully* a resource is by a GPU kernel, i.e., the boundedness of resource utilization. The level of component use is a good metric for measuring the imbalance of resource utilization in a GPU program kernel and therefore provides guidelines of which kernels might be good candidates for merging.

The second contribution is the kernel merging transformation that provides multiple ways of merging GPU kernels to balance the resource usage on GPU. We also developed several heuristics to guide the transformation to maximize benefit. The third contribution is that the proposed transformation is, to our best knowledge, the first cross-platform code transformation technique that addresses the resource under-utilization problem for both AMD's and NVIDIA's GPGPU programming frameworks.

The rest of this paper is organized as follows. Section 2 introduces the architectural features of AMD and NVIDIA GPUs that are relevant to the code merging transformation. Section 3 explains from a GPU architecture point-of-view why code merging might improve program performance on GPU. Section 4 describes the transformation and heuristics of how and when to apply the transformation. Then, we comprehensively evaluate our approach on both AMD and NVIDIA GPUs and show the results in Section 5. Finally we conclude and suggest future directions of research in Section 6.

2 GPU Background

Modern GPUs are massively multi-threaded many-core architectures with several layers of memory in a complex memory hierarchy scheme. One key aspect in the evaluation of the performance of GPU programs is the bottleneck, which is the limiting factor of the application. There are essentially two types of bottlenecks on GPUs: ALU and memory. An ALU bound GPU kernel is a kernel in which the majority of the time executing the kernel is spent doing ALU operations. A memory bound GPU kernel is a kernel in which the majority of the time executing the kernel is spent doing memory operations. The GPU achieves the greatest utilization when both ALU and memory are fully used.

The two main brand name GPU architectures share similar features but are also quite different. The terminologies are also different on the two architectures even for features that work in similar ways. In the following we briefly introduce the architectural features on NVIDIA and AMD GPUs that are most relevant to the proposed code transformation. We will refer the detailed introduction of GPU programming on the two architectures to their respective programming guides.

2.1 AMD GPU

The AMD GPU consists of multiple compute units (SIMD engines) each of which has 16 stream cores (thread processors). Each stream core itself contains processing elements that make up a 5-wide VLIW processor. The memory hierarchy consists of special registers, global purpose registers, L1 and L2 cache, local memory, texture and vertex fetch and global memory. Threads are organized into work-items and then into wavefronts. For the high end AMD compute device, a wavefront consists of 64 threads organized into 16 2x2 quads, and wavefronts are then organized into work-groups. A compute device has two wavefront slots, odd and even, and executes two wavefronts simultaneously. More wavefronts can be in the work queue and can be switched with an executing wavefront when the executing wavefront stalls. The number of wavefronts that can exist in the queue is dependent on the number of resources available, such as registers and local memory. This is an important feature because this wavefront switching allows for greater GPU utilization and gives the GPU the ability to hide memory latencies. The instruction set architecture is organized into VLIW instructions or bundles. These bundles are then organized into clauses, which are a group of bundles of the same type. For example, ALU operations are grouped into an ALU clause while fetch operations might be grouped into a TEX clause. Each clause is executed by a wavefront until completion and cannot be switched out mid-clause.[2]

2.2 NVIDIA GPU

The NVIDIA GPU consists of multiple stream multiprocessors with 8 streaming processors per multiprocessor. The memory hierarchy consists of registers, shared memory, texture memory, constant memory and global memory. Threads are organized into groups of 32, called warps. Each warp is executed in SIMD fashion on a stream multiprocessor and is broken into half-warps, 16 threads, and quarter-warps, 8 threads.

Like the switching between clauses on AMD GPUs, NVIDIA GPUs also allow switching between warps when warps stall. Warps are organized into blocks which are then organized into grids. While NVIDIA exposes PTX (an intermediate representation of instructions) they do not currently expose their ISA directly. Since PTX is an intermediate representation and is not directly executed on GPU hardware, it is not able to give the same level of detail as an instruction set architecture.[1]

3 Motivation

The main motivation behind this code merging transformation is the idea to improve performance by balancing the utilization of GPU resources within kernels. In general, most current GPU applications are implemented with the goal to get maximum ALU utilization. While the ideal full ALU utilization may give the best performance for an application, some algorithms are naturally memory bound and simply don't allow their GPU implementations to become ALU bound. These memory bound GPU kernels leave many ALU resources idle. In the same notion, ALU bound GPU kernels leave many memory resources idle. We note that GPU is an intrinsically throughput-oriented architecture. Our proposed transformation attempts to achieve better overall GPU utilization by recognizing idleness, both ALU and memory bound, and combining GPU kernels that are on different ends of the ALU/memory usage spectrum so as to move the point of usage of the combined code closer to neutral which implies better utilization of GPU resources and better overall performance. Equally important is the criteria that determines which kernels exhibit good qualities of a merging candidate. This type of transformation would be particularly helpful for a multi-user GPU system, because the proposed transformation can improve the overall throughput by recouping idle resource capacity across different applications that run at the same time on such shared systems.

We want to address the relationship between the code merging transformation with other optimization techniques on the GPU. The proposed transformation is compatible with most other GPU optimizations. The transformation occurs at a coarser granularity than other known optimizations, most of which occur within the kernel itself such as memory and register optimizations[3][4] and divergence and workload balance optimizations[5][6]. This transformation does not interfere with these known optimizations because these optimizations can still be applied after the proposed transformation to the merged kernels. There also exist some optimizations which require the kernel code to be split into multiple kernels[7]. This transformation does not affect this type of optimization either since it can still be applied to the split kernels, if there are other kernels to merge with them. Furthermore, this transformation can also be applied with emerging GPU technology such as concurrent kernels [1] since several kernels can be merged into one or more kernels and can then be run concurrently under the same context.

The discussion of the level of use of resource utilization provides a high-level view of why the proposed code merging transformation works. Next we go into the design details of NVIDIA's and AMD's GPU to reveal an architectural explanation of how code merging, if done properly, leads to a better utilization of the ALU/memory units and improves program performance on GPU.

Furthermore, the second major motivation behind this transformation is to merge kernels from different processes on multi-user systems into one process. For example, if a developer is utilizing a system for executing an ALU intense algorithm and another developer is waiting to execute a memory intense algorithm, these two processes' kernels can be merged into one larger kernel resulting in better overall GPU utilization and kernel speedup. Thereby, this type of transformation would be particularly helpful for a multi-user system.

Next we discuss three architectural/program features that are not ideally handled in current technology and can be improved by the proposed transformations.

3.1 ALU Packing Percentage

Since NVIDIA GPUs do not utilize VLIW processors this metric applies only to AMD GPUs. The ALU packing percentage is the percent of cores in the VLIW processor that are being utilized by the GPU kernel. For example, if the ALU packing percentage is 20% then only 1 of the 5 cores are being used per VLIW instruction. There are two major factors that affect the ALU packing percentage. The first factor is that of data dependence. Instructions that have data dependence are not able to be scheduled together in one VLIW instruction. The second factor is that of control flow and scope. Instructions within a control flow statement or scope are scheduled, by the compiler, to run in separate ISA clauses. Instruction packing does not occur across clauses and so while there may not be any data dependence between an instruction outside an if statement and one inside an if statement, these two instructions cannot be packed. The ALU packing percentage can be increased through this transformation because separate GPU kernels have no data dependence and their instructions can be bundled together within VLIW instructions. Instructions within control flow statements can only be combined across GPU kernels if the control flow statements have the same conditionals or the instructions within the control flow statements can be software predicated or the two kernels have synchronization points which can be combined. Figure 1 and Figure 2 both have data dependence from instruction 13 to 14 in ALU clause 02 and are only utilizing 1 of the 5 cores in each VLIW instruction. If these two kernels are merged, the resulting code, as shown in Figure 3 now has 2 of the 5 cores in each VLIW instruction being utilized. Basically the merged code uses the same number of cycles as either of the pre-merge kernels, as if one is piggy-backing the other.

3.2 Idleness

There are three major GPU components which can execute in parallel: ALU components, global memory components and local memory components. Kernel code is generally laid out in a streaming fashion: read input, work on input and write output. This inherent layout causes dependence to occur between the components and while there can exist some overlap in resource usage due to context switching, there still leaves idleness in those components in which the kernel does not greatly use. For example, in an ALU bound kernel each wavefront uses the ALU units far more than the memory units and the time it takes to execute one wavefronts' ALU operations can be spent

```
02 ALU: ADDR(51) CNT(117)
13 w: ADD T0.w, T0.y, PV12.x
14 z: ADD T0.z, T0.x, PV13.w
```

Fig. 1. Kernel One ISA

```
02 ALU: ADDR(51) CNT(63)
 13 y: ADD T0.y, T0.w, PV12.z
 14 x: ADD T0.x, T0.z, PV13.y
```

Fig. 2. Kernel Two ISA

```
02 ALU: ADDR(56) CNT(121)
 13 x: ADD T0.x, PV12.w, R2.x
    z: ADD T0.z, R0.x, PV12.w
 14 y: ADD T0.y, T0.w, PV13.z
    w: ADD T0.w, T0.w, PV13.x
```

Fig. 3. Merged Kernels ISA

fetching inputs for multiple wavefronts from memory. This leaves a gap when executing the back half of the wavefronts such that all the inputs have been fetched for all the wavefronts but not all the ALU operations have been executed. In this case, there is an opportunity to use the memory units that are sitting idle. In contrast, in a memory bound kernel the memory units are used far more than the ALU operations and therefore, in the same sense, the ALU units are not being fed fast enough and are sitting idle. There also exists the case where local memory is being used but neither the ALU units nor the global memory units are being used. In both types of memory, global and local, stalling is considering as being busy since the unit is being tasked. Stalling is not the same as idling. In this paper, these are the three major cases that motivate this kernel merging technique.

3.3 API Overhead

Each process or application requires a certain amount of API overhead or setup time. This not only includes the individual kernel invocation time but also count the entire API overhead and setup time, such as the setting up of device and the creation of context. For individual processes each of these steps must be done at least once and while kernel invocation time can be reduced with a single application by queuing the kernels it cannot be reduced when looking at kernels across processes. When kernels are merged this overhead of setup is reduced since these calls only need to take place once. This overhead reduction scales with the number of merged kernels across programs since the device, context and platform setup times remains the same.

4 Transformation

Kernels from different processes are merged in two steps. The first step brings the two separate host codes into one program and the second step is to merge the kernel codes. The main work in the first step is to create a proper context for the kernel merging in the second step. The context preparation involves merging the data structures, functions and necessary code into one program. Furthermore, we also bring the two separate OpenCL

API calls under one ”umbrella” of OpenCL API calls, i.e., setting up the platform, device, context and kernel for OpenCL. In addition, all of the buffer creation, buffer reads, buffer writes and releasing objects need to be merged. The first step may appear to involve a lot of things, however, is actually very straightforward because it’s not important in which order these things get merged in context preparation, so long as they stay in the proper order within the code, i.e. creating buffers before writes and writes before kernel call and kernel call before reads and reads before releases. The performance of the merged kernel is not affected either as long as the context is prepared in a valid way.

Merging the kernel code in the second step is not as straightforward and the way in which the kernel code is merged can greatly impact performance. The simplest way to merge two kernels is to cascade the kernel codes as shown in Figure 4. However, simple solutions like cascading do not usually give optimal performance. Here we will discuss several key factors that are important to the decision of how to merge. Furthermore, we describe several heuristics for choosing which kernels to merge. The heuristics are very effective and give near optimal performance in most cases.

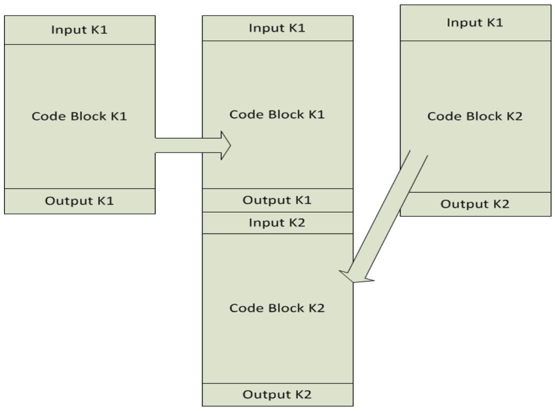

Fig. 4. Cascading Kernels

4.1 Merging Based on Resource Usage

Generally speaking, kernels that use resources in different ways can potentially benefit from merging. There are two values which impact this decision: actual resource busy percentages and theoretical resource busy percentages. Both of these values give an indication of the bottleneck, the profiler giving the real bottleneck and the theoretical values giving what ”should” be the bottleneck. The theoretical values can also be looked at as a best case scenario (components are totally parallelized, so the execution time is the greatest of the three components’ times) or a worst case scenario (components are totally serialized, so the execution time is the sum of the three components’ times). In reality, neither of the theoretical times are true since there is dependence and also some parallelization of work. However, these values can be used as a guide when compared

Table 1. AMD OpenCL Profiler Counters

Counters	Description
Time	Kernel execution time not including kernel setup
ALU Instr	Number of ALU instructions
Fetch Instr	Number of global read instructions
Write Instr	Number of global write instructions
LDS Fetch Instr	Number of LDS read instructions
LDS Write Instr	Number of LDS write instructions
ALU Busy	Percent of overall time ALU units are busy
Fetch Busy	Percent of overall time read units are busy (includes stalled)
Write Stalled	Percent of overall time Write units are stalled
LDS Bank Conflict	Percent of overall time LDS units are stalled by bank conflicts
Fast Path	Total KBs written using the fast path
Complete Path	Total KBs written using the complete path

to the profiler values as to how much serialization, parallelization or idleness is actually occurring in a kernel. In this paper, we use these two sets of values of kernels to decide the potential benefit for merging them.

For this transformation, the profiler counters used are described in Table 1. Some of the counters are used to calculate the actual run times of the components and some of the other counters are needed to calculate the theoretical values. Equations 1 and 2 refer to the equations to calculate the theoretical time spent on each component. Equations 3, 4, 5 and 6 are the equations to calculate the actual time spent on each component. The actual time spent on ALU and read operations is easily calculated since the profiler directly reports the percent time the kernel is busy performing these operations. The actual time spent using the LDS (local memory) and writing data to global memory is given as an estimation using some profiler counters together with the theoretical equations since the profiler does not report direct busy percents on these operations.

$$\text{Global Memory Time} = \frac{\text{Total Bits}}{\text{Bus Width} \times \text{Memory Clock}} \tag{1}$$

$$\text{LDS Time} = \frac{\text{\# Threads} \times \text{LDS Instr}}{\text{LDS per Clock} \times \text{Engine Clock}} \tag{2}$$

$$\text{Actual ALU Time} = \text{Kernel Time} \times \frac{\text{ALU Busy}}{100} \tag{3}$$

$$\text{Actual Read Time} = \text{Kernel Time} \times \frac{\text{Fetch Busy}}{100} \tag{4}$$

$$\text{Est. Actual Write Time} = (1) + \text{Kernel Time} \times \text{Write Stalled} \tag{5}$$

$$\text{Est. Actual LDS Time} = (2) + \text{Kernel Time} \times \text{LDS Conflict} \tag{6}$$

For example, given a kernel with 98% ALU Busy and 22% Fetch (read) Busy counters we can determine that there is 1) overlap in ALU workload and global memory read workload (since together they are greater than 100%) and 2) 78% of the time the memory read units are idle. Since this kernel is 98% ALU Busy this would indicate that this kernel would not be a good candidate for merging with another ALU bound kernel based on resource usage. This kernel is however only using the read units 22% of the time and therefore would make a good candidate for merging with a read bound kernel (since although adding ALU operations would increase the ALU execution time in an already ALU bound kernel, it would hide memory latency from the second kernel). A kernel with the opposite values, 98% Fetch Busy and 22% ALU Busy would be a good candidate for merging with an ALU bound kernel (since the first kernel could hide some ALU operations of the second kernel).

The resource usage of a kernel is represented as a vector of five values: an ALU value, a global read value, a global write value, an LDS value and an overall ”overlap” value. The vectors will be used as the decision factor for the purposes of choosing which kernels to merge and the way of merging. The first four values represent how much of the resource is being used by the kernel and the last value, the overlap value, represents how much of that work is being done in parallel by the components. The ALU value and global read value come directly from the counters ALU Busy and Fetch Busy, respectively. The global write value and LDS value are the percents of equation 5 and equation 6 in reference to the Time counter, respectively, since the profiler does not directly give the percent these units are busy. The overlap value is important because it gives detail as to how much resource availability within that time exist among the different components and how much room in each component is available for merging. In the above example, if the overlap is 0 then the ALU is idle during the entire fetch time and vice-versa. A kernel with a low overlap value makes for a better merge candidate than a kernel with a higher overlap value. If a kernel is 98% ALU busy and 50% fetch busy and has an overlap of 100% then there is only 50% of the total kernel time to do extra fetch instructions.

The individual actual times are calculated and summed and the percent difference is taken with regards to the Time counter. This calculation yields the overlap value equation below:

$$\text{Overlap} = 1 - \frac{\text{Kernel Time}}{(3) + (4) + (5) + (6)} \qquad (7)$$

This equation is not a direct calculation for the actual overlap since it is not currently possible to directly extract the level of component parallelization from the profiler that has no such counter. Unfortunately, at this time, it is not possible to tell which components' runtimes overlap with which other components' runtimes, making it not possible to give an exact formula for estimating benefit based on resource usage. Instead, this heuristic is used to give a quantifiable estimation of the overall overlap.

Each kernel candidate's values, the five listed above, are summed together. For example, given two kernels with the above properties would result in an ALU value of .98 + .98 = 1.96 and a global read value of .22 + .22 = .44, so these would not be good merging candidates because both kernels are very ALU bound. For example, if we assume that these two kernels serialize the read and ALU operations (no overlap) then only a speedup of the time of one kernel's read operations can be hidden. However, given one kernel with the properties mentioned above and one with opposite properties the ALU value would be .98 + .22 = 1.20 and a global read value of .22 + .98 = 1.20, giving a much better balance and would hence be good merging candidates since the fetch bound kernel could hide memory latency in the ALu bound kernel. For example, if we again assume that these two kernels serialize the read and ALU operations then the speedup could be .98 - .22 = .68 of the second kernels' read time. What this means is that if the first kernel spends 22% of it's time reading data then 98% of it's time executing ALU operations, the second kernel can spend 68% of the first kernel's time reading data. Speedup depends on the level of overlap for each kernel, which in practice is neither 0% or 100%, even in the first example the ALU is not busy 100% of the time.

4.2 Merging Based on ALU Packing Percentage

Another attribute that contributes to the merging decision is the ALU packing percentages. This heuristic only applies to AMD's VLIW architecture and, unlike merging based on resource usage, would allow two ALU bound kernels to be merged and obtain speedup. When merging kernels, one kernel's ALU instructions might fit into the empty VLIW slots of the other kernel, thereby reducing the overall number of ALU cycles needed to execute the kernel. The compiler has control of how the VLIW instructions are packed and the size of the window the compiler uses when looking at which VLIW instructions to pack. The window for packing analysis is between the point in which two code blocks meet. In other words, the compiler's window for packing can only reach so far up and down and won't pack a merged kernels' instructions if they are outside of that window. Greater code motion and interleaving of code statements between the merged kernels could lead to better packing improvements but would, in most instances, not overcome the negative effects of the increased register usage. It is still possible to get speedup from this feature with those constraints. For example, given two ALU bound kernels with 500 ALU instructions and 440 instructions merged over 2048*2048 threads, a speedup of 1.13 is obtained on AMD's Radeon 5870. The decision to merge based on ALU packing percentage should be evaluated after the kernel candidates have failed the test based on resource usage. Specifically, the ALU packing heuristic is based on 1) the kernel's packing percentage (lower is better) and 2) the kernel's ALU usage (higher is better). One of the other advantages to ALU packing is that profile feedback is not needed, ALU packing can be determined statically through the analysis of the assembly code of the program.

4.3 General Techniques of Kernel Merging

When merging kernels there are some good practices that generally adhere to most scenarios. One key issue when merging kernels is the impact the extra code will have on register pressure. Register pressure is always a performance consideration when programming for the GPU and, generally, the lower register pressure the better since more threads can be queued which leads to more memory latency hiding. Ideally, since the kernels are independent pieces of code, there should be little to no added register pressure. The register count used in the merged code should be no more than the highest register count among all kernels. This is true when the kernels are simply cascaded because all the registers used by the top kernel can then be reused by the bottom kernel, so this technique does not significantly impact register pressure. There is no need to have both sets of registers reserved because registers can be reused from one kernel's code to the next when the kernels are merged. However, simply cascading the kernels is not always optimal for performance. Every kernel has some output and when merging kernels are simply cascaded the output of one kernel is put in the middle of the newly merged kernel code. In most instances this causes the compiler to produce a wait acknowledgment assembly instruction $(WAIT_ACK)$, which adds synchronization to the kernel and causes slowdown because it waits for acknowledgments back from either the read or write memory units. To remove this synchronization, the output of all the merged kernels are moved toward the end of the kernel as shown in Figure 5. In this way, the register pressure is not significantly affected. It's not always possible to put the outputs right next to each other for the kernels being merged, instead a best effort is made while keeping register pressure as low as possible (without reducing occupancy) and eliminating the $(WAIT_ACK)$ instruction. Similar to the principle of maximizing

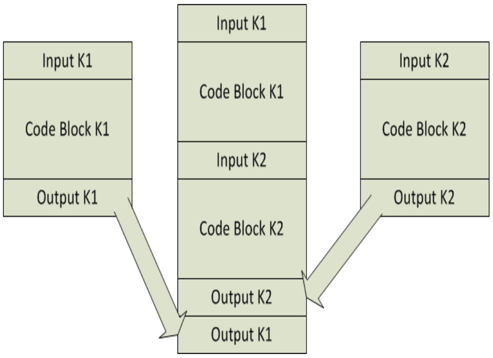

Fig. 5. Moving Output of Merged Kernels

the grouping of outputs, every attempt should also be made to group the inputs. However, this is not due to a wait acknowledgment after a memory read but is driven by the goal to group the memory reads in the same memory read clause, so as to reduce any delay caused by switching between two different such clauses or any delay caused by data dependence. Since the effect of this optimization might not be high, it is not performed if the register count needs to be increased to the point of reducing the number of wavefronts/warps. This is shown in Figure 6. For AMD GPUs, the equation to calculate the number of simultaneous wavefronts is:

$$\text{Simul WFs} = \frac{\text{Regs per Thread} - \text{Temp Regs used}}{\text{Regs Used}} \tag{8}$$

4.4 Fences and Barriers

There are many types of kernels which require synchronization and in these kernels fences and barriers are used to implement this synchronization. There are two cases to consider when merging kernels with synchronization primitives: both kernels have synchronization or one kernel has synchronization. In the case of both kernels having

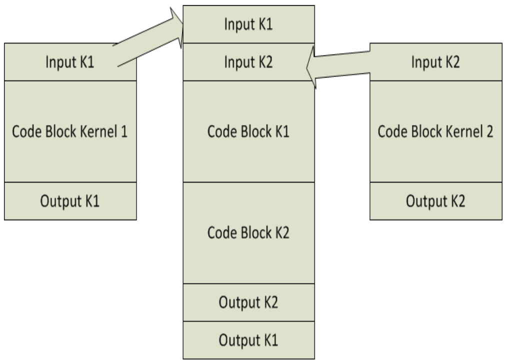

Fig. 6. Moving Input of Merged Kernels

synchronization each kernel code is broken into blocks separated by these fences or barriers. The blocks are then merged based on their sequence according to the blocks as shown in Figure 7. The case of merging kernels with synchronization primitives is a good example showing that a simple merging approach such as cascading can sometimes destroy performance. If the kernels were simply cascaded the total number of fences or barriers would be the sum of the synchronizations of all the kernels being merged. However, merging kernels at the granularity of code blocks that are synchronized allows the number of synchronizations to remain the same as the kernel with the highest number of synchronizations because the kernels now share the same barriers, thus eliminating the need for duplicate barriers. For example, if each kernel performs some operations and then synchronizes, in the merged kernel they can both perform these operations at the same time and then synchronize together.

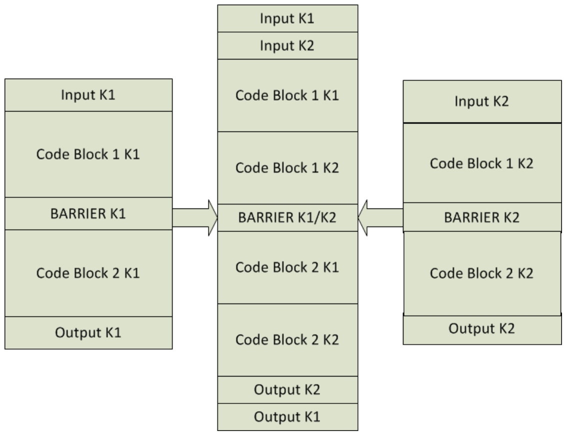

Fig. 7. Merging Kernels with Synchronization Points

If only one kernel has synchronization operations, the kernel that does not have synchronization is placed either before or after the synchronization operation depending on which option gives the lowest register usage, which can be determined statically. This technique does not exclude the other techniques mentioned above, meaning that both moving the output and inputs is taken in to account when merging kernels with synchronization.

4.5 Limiting Factors

The proposed kernel merging technique cannot be employed freely. There are a few other limiting factors that should be considered when deciding whether to apply this transformation or not. One limitation is caused by the possible changes of the dimensions of a GPU kernel when it is merged with another kernel, and the consequent changes in its memory access patterns. For example, changing the local dimension from 1D to 2D, or vice versa, could cause global memory conflicts or contention. Kernels work best when they execute over the dimensions, both global and local, for which they were coded. For some applications a specific local thread size and/or dimension are needed for correct results, particularly for problems that utilize a certain block size for local memory calculations. For example, matrix operations often fall into this category. There are also applications that may have their performance effected by the local dimension size when going from one dimension to larger dimensions due to a change in memory access size based on the algorithm. For example, image filters often look at the surrounding pixels and the size of the reads can change depending on the local dimension, whether they are just looking forward and backward or forward, backward, side to side and diagonally.

Our heuristic considers both local memory usage and local thread size. If a kernel has a two dimension local thread size (block size) it will not be reduced when merging. What this means is that if this kernel is merged with a one dimension local thread size kernel, that kernel's local thread size is changed to two dimensions, unless changing the local thread size effects the fetch size performance (architecture dependent). Our heuristic checks to make sure that the combined local memory usage does not exceed the local memory available for the architecture, if it does then those kernels are not merged.

5 Evaluation

We evaluate our code merging transformation technique on both AMD and NVIDIA GPUs. All of the results presented in this paper were taken with maximum global size and maximum memory usage allowed for the given problem, thereby imitating a large data inter-process environment. The host-to-device transfer will not be addressed in our evaluation since it does not impact the effectiveness of the proposed transformation in one way or the other. The AMD GPU used was the 5870 which has the RV870 chip and 1GB of memory. The NVIDIA GPU used was the GTX280 which has the GT200 chip and 1GB of memory. The specifications of both GPU architectures are mentioned in the GPU Background section of this paper. The latest drivers and SDKs were used at the time of this writing, which was the AMD Stream SDK 2.2 and NVIDIA's CUDA/OpenCL 3.1 Toolkit. All profiling information listed was obtained using the AMD OpenCL profiler.

The benchmarks used for evaluation were taken from the AMD Stream SDK 2.2 OpenCL samples. The code is compatible with but might not be tuned for NVIDIA hardware. However, since we use the same compiler options before and after our transformation, the speedup validly reflect the effectiveness of our approach. The runtimes are taken directly from these samples, unaltered for either GPU architecture. The merged kernels were merged without modifying any kernel code other than the transformation techniques outlined in the Transformation section of this paper. All of the speedups shown were compared to the combined single benchmark times using the same global size with an optimal local size. The benchmarks are one-dimension BlackScholes, Mersenne Twister, SobelFilter and SimpleConvolution, as well as two-dimension DCT and MatrixTranspose. The six benchmarks can be combined into 15 different pairs (C_6^2), and we tested all combinations that passed the heuristic tests.

The results for both the AMD 5870 and the NVIDIA GTX280 are listed in Figures 8 and 9, respectively. All the timing results were obtained using the respective profilers: AMD's Stream OpenCL profiler and NVIDIA's OpenCL profiler. The AMD GPU shows the most significant performance gain with an average speedup of 1.28 and a maximum speedup of 1.53. Good speedups were also obtained for the GTX280 (average speedup 1.17 and max 1.37).

5.1 Effects of Resource Usage

Barring any of the mentioned limiting factors, the best performance results come from the merging of ALU bound kernels with memory bound kernels. This conforms to our transformation heuristics and allows for the greatest increase in overlap. For our benchmarks, the DCT, MatrixTranspose and SimpleConvolution all had relatively low ALU values and overlap values while the BlackScholes, SobelFilter and MersenneTwister samples had relatively high ALU and overlap values, as can be seen in Table 2.

Table 2. Profiled Resource Utilization of Single Benchmark

Kernel	ALU	Read	Write	LDS	Overlap
DCT	0.381	0.224	0.016	0.429	0.049
Black Scholes	0.997	0.222	0.110	0	0.241
Mersenne Twister	0.956	0.086	0.130	0.280	0.311
MaxTrans	0.025	0.006	0.010	0.057	-9.06
Simple Conv	0.434	0.334	0.026	0	-0.25
Sobel Filter	0.960	0.231	0.035	0	0.184

In Table 2 the kernels with a high ALU value also have a high overlap value since these kernels are ALU bound and, combined with context switching, allows for a high amount of memory latency hiding. This means that these kernels will merge well. The DCT sample is LDS bound and therefore has little overlap and will merge well with a kernel that has either a high ALU value, a high read value or a high write value. Both the Matrix Transpose and the Simple Convolution benchmarks show a negative overlap, however. The profiler does not give the information about all aspects of execution that might impact performance, such as unreported bank conflicts, LDS stalls, synchronization stalls, and context switching latency. However, for the purpose of merging, since we know the negative overlap is not caused by the ALU or read values, the negative value itself indicates that there is heavy idleness.

The values are added to predict which kernels are good merge candidates. For example, if two kernels are to be merged the speedup gained is going to decrease as any value approaches 2. For example, for any combination of Black Scholes, Mersenne Twister or Sobel Filter on NVIDIA 280 GTX, the sum of their ALU values readily approaches the threshold and so the speedup is nominal. The ALU value for Matrix Transpose and Black Scholes is just over 1, meaning that most of the other operations can be hidden by ALU operations while being just slightly more ALU bound. This method is done for each of the first four values with the same concept. For example, if two kernels had read values of .99 then they wouldn't be good merge candidates. The best merge candidates are going to stress different components of the GPU, ideally a perfectly merged kernel with have values of 1. This can be extended to multiple kernels (more than two) since there are more than two components that work in parallel and most kernels won't use close to 100% of any of those components. For example, a high ALU value kernel could be very well merged with a high read value kernel and a high LDS value kernel. This can be extended to as many kernels as the heuristic will predict speedup.

Looking at Table 3, we can see that the Matrix Transpose kernel merges well with almost every other kernel since it's values are so small and it's overlap is deeply negative, there's plenty of opportunity for other kernels to do work during it's runtime. This is expressed in Table 3 by the improvement in the overlap value.

Table 3. Heuristics values for kernel combinations

Merged Kernel	**ALU**	**Read**	**Write**	**LDS**	**Overlap**
DCT+Twister	0.770	0.141	0.104	0.370	0.278
DCT+Scholes	0.803	0.258	0.085	0.240	0.278
Scholes+MaxTrans	0.810	0.185	0.114	0.071	0.153
Scholes+Twister	0.978	0.139	0.144	0.206	0.318
Twister+MaxTrans	0.765	0.072	0.125	0.297	0.206
DCT+MaxTrans	0.219	0.128	0.018	0.294	-0.514
Scholes+Simple	0.854	0.263	0.098	0	0.177
Twister+Simple	0.836	0.138	0.122	0.248	0.257
DCT+Simple	0.351	0.217	0.018	0.233	-0.217
Simple+MaxTrans	0.122	0.077	0.013	0.037	-2.98
Scholes+Sobel	0.998	0.223	0.109	0	0.249
Twister+Sobel	0.940	0.106	0.131	0.266	0.308
DCT+Sobel	0.442	0.182	0.021	0.279	-0.079
Simple+Sobel	0.683	0.297	0.042	0	0.0227
Sobel+MaxTrans	0.131	0.029	0.013	0.038	-3.68

5.2 The Effects of Barriers

Of the 6 benchmarks only the DCT and MatrixTranspose have barriers. Each of the two has one barrier. These two kernels are merged according to the Fences and Barriers subsection of the Transformation section. That is, the code segments that are separated by barriers are interleaved in the merged kernel. When merging the DCT and Matrix-Transpose, the code sections on each side of the barrier were put on the proper side of the barrier in the merged kernels, so no extra barriers were needed. Despite the need for synchronization, these samples give good performance on the AMD GPU when merged with BlackScholes and MersenneTwister, both of whom are ALU bound. DCT and MatrixTranspose also show speedup when merged together, both due to Matrix Transpose's low overlap value and interleaving the barriers.

5.3 Candidate Elimination

Table 3 shows all of the values for all possible combinations and indicates which combinations are good candidates for merging based on our heuristics. Both the Sobel Filter and Simple Convolution cause an increase in fetch size when going from a local thread dimension size of one to two on AMD's Radeon 5870 and the Sobel Filter on the GTX280. The increase in runtime is greater than the expected benefit from the transformation so the kernels aren't merged when it requires that they be transformed into two dimensional problems. Additionally for the NVIDIA GPU, the combinations that were eliminated were Scholes Twister, Scholes Sobel and Sobel Twister. All of these kernels have high ALU values and since the NVIDIA GPU doesn't use the VLIW architecture there is no possible gain from ALU packing. This causes their runtimes to be serialized and no improvement can be made. These three kernels might also be eliminated from the AMD GPU if not for the possible gain in performance from an increase in ALU packing. The advantage of ALU packing does not need to be profiled and can be attained statically through the compiled ISA. The speedups from these combinations of ALU bound kernels on the AMD GPU comes from ALU packing. The number of ALU cycles saved in the Scholes Twister sample was about double that of the Sobel Twister and Sobel Scholes samples. Figures 8 and 9 show the results of all the combinations that pass the heuristics and are actually merged. The left column in all the figures shows the execution time of kernels executed separately, and the right column shows the execution time of the corresponding merged kernel.

Just for the purpose to show the effectiveness of our heuristics, Figure 10 shows all of the results for the combinations of benchmarks that did not pass our heuristics. The combinations in this figure have almost the same execution time for the merged kernel as the combined execution time of the individual kernels. On the other hand, it also shows that our transformation is "safe" in the sense that even if two wrong kernels are merged, the performance will only marginally decrease.

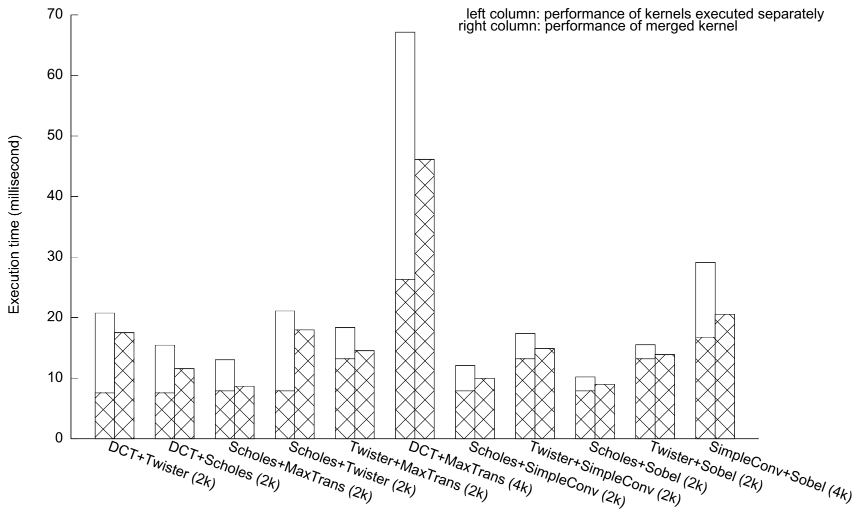

Fig. 8. AMD 5870 Results

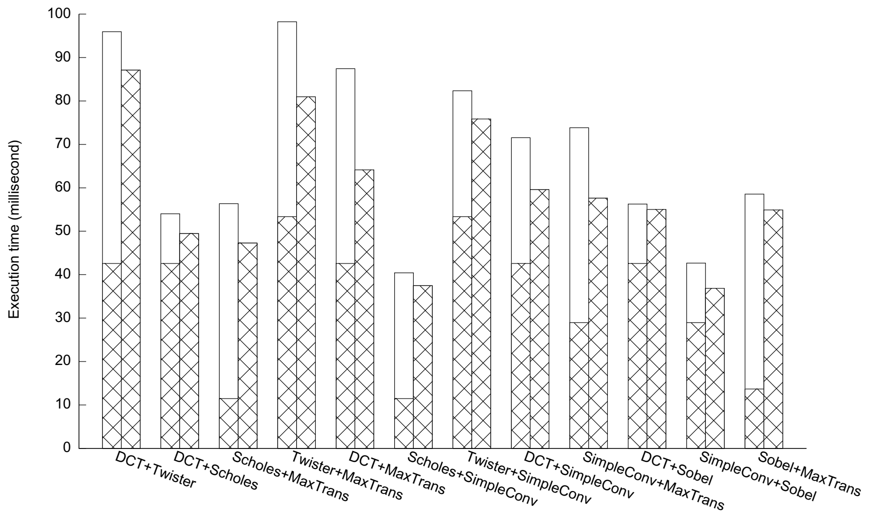

Fig. 9. NVIDIA GTX280 Results

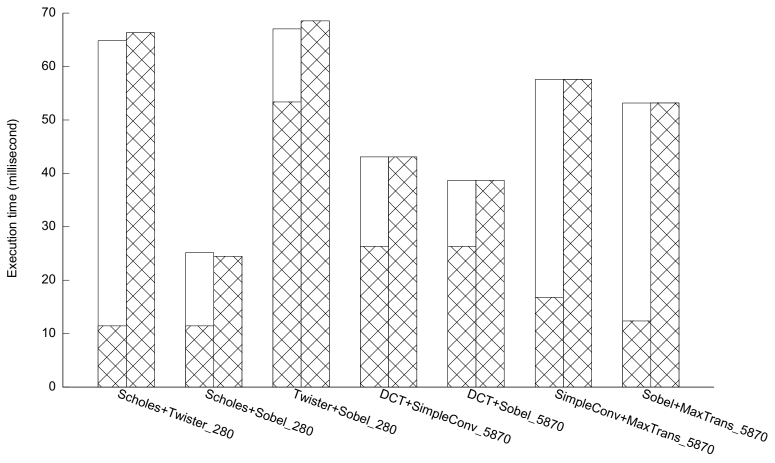

Fig. 10. Results of Poor Merging Candidates

6 Related Work and Conclusion

The performance effect of resource sharing and competition has been studied before in the context of multi-threaded CPU programs. Resources in a CPU including cache, off-chip bandwidth and memory are shared among many multi-tasked threads. A preview of the phenomena of inter-thread conflict and thrashing is already observed in multi-threaded machines such as the Intel Pentium 4 that supports hyperthreading [8,9], even though there are only two threads that share a processor and other resources. [10,11,12] explores the possibility of optimizing multiple programs that will run simultaneously on multi-processors. Generally, the previous approaches are designed for multi-threaded programs with a small number of threads and cannot be easily scaled up to hundreds of threads that run in parallel on a GPU. Intra-thread optimizations for GPU have been long studied. There are many well documented optimizations for GPU programs that are applied at the thread or wavefront/warp level to improve the usage of GPU resources. A recent example is a systematic framework to optimize for GPU memory hierarchy that is proposed in [13]. A good overview of intra-thread optimizations can be found in [14]. On the other hand, however, little is known about optimizations for resources at a higher level such as at the kernel level.

In this paper we present a novel transformation that merges inter-process kernels to improve program performance on the GPU. The proposed transformation is motivated by the fact that while there are many techniques that optimize GPU kernels at the thread level there are no techniques that help to increase resource utilization below the kernel level. Concurrent kernels, while improving overall GPU usage by utilizing more resource units at one time, does not help to improve individual resource units' usage within the GPU. We discuss how the code merging transformation has minimal effect on optimizations done at a finer granularity and can help increase performance when combined with other optimizations. We give a set of heuristics for selecting which kernels to merge and how to merge them along with real world application results from both major GPU vendors for a wide array of benchmarks.

In the future, we would like to extend this work to include larger benchmark applications that contain multiple kernels. We would also like to extend this work to include results for Fermi machines and for the new 4-wide VLIW AMD machines, for both the set of benchmarks presented and the planned future larger benchmark applications. A comparison between the speedups of this technique and the speedups from using concurrent kernel execution will also be included.

References

1. Nvidia OpenCL Programming Guide (May 2010)
2. AMD OpenCL Programming Guide (June 2010)
3. Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., Hwu, W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 73–82. ACM (2008)
4. Ueng, S.-Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.-M.W.: CUDA-Lite: Reducing GPU Programming Complexity. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 1–15. Springer, Heidelberg (2008)
5. Zhang, E., Jiang, Y., Guo, Z., Shen, X.: Streamlining GPU applications on the fly. In: Proceedings of the 24th ACM International Conference on Supercomputing, pp. 115–126. ACM (2010)
6. Chen, L., Villa, O., Krishnamoorthy, S., Ga, G.: Dynamic load balancing on single- and multi-GPU systems. In: 2010 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–12. IEEE (2010)
7. Carrillo, S., Siegel, J., Li, X.: A control-structure splitting optimization for GPGPU. In: Proceedings of the 6th ACM Conference on Computing Frontiers, pp. 147–150. ACM (2009)
8. Hily, S., Seznec, A.: Contention on 2nd Level Cache May Limit the Effectiveness of Simultaneous Multithreading. Technical Report PI-1086 (1997)
9. Leng, T., Ali, R., Hsieh, J., Mashayekhi, V., Rooholamini, R.: A Study of Hyper-threading in High-performance Computing Clusters. In: Dell Power Solutions HPC Cluster Environment, pp. 33–36 (2002)
10. Kandemir, M.: Compiler-Directed Collective-I/O. IEEE Trans. Parallel Distrib. Syst. 12(12), 1318–1331 (2001)
11. Hom, J., Kremer, U.: Inter-program Optimizations for Conserving Disk Energy. In: ISLPED 2005: Proceedings of the 2005 International Symposium on Low Power Electronics and Design, pp. 335–338. ACM Press, New York (2005)
12. Ozturk, O., Chen, G., Kandemir, M.: Multi-compilation: Capturing Interactions Among Concurrently-executing Applications. In: CF 2006: Proceedings of the 3rd Conference on Computing Frontiers, pp. 157–170. ACM Press, New York (2006)
13. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 86–97. ACM (2010)
14. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.Z., Baghsorkhi, S.S., Hwu, W.W.: Program optimization carving for GPU computing. Journal of Parallel and Distributed Computing 68(10), 1389–1401 (2008)