Sponge: Portable Stream Programming on Graphics Engines

Amir Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{hormati, mehrzads, mwoh, tnm, mahlke}@umich.edu

Abstract

Graphics processing units (GPUs) provide a low cost platform for accelerating high performance computations. The introduction of new programming languages, such as CUDA and OpenCL, makes GPU programming attractive to a wide variety of programmers. However, programming GPUs is still a cumbersome task for two primary reasons: tedious performance optimizations and lack of portability. First, optimizing an algorithm for a specific GPU is a time-consuming task that requires a thorough understanding of both the algorithm and the underlying hardware. Unoptimized CUDA programs typically only achieve a small fraction of the peak GPU performance. Second, GPU code lacks efficient portability as code written for one GPU can be inefficient when executed on another. Moving code from one GPU to another while maintaining the desired performance is a non-trivial task often requiring significant modifications to account for the hardware differences. In this work, we propose Sponge, a compilation framework for GPUs using synchronous data flow streaming languages. Sponge is capable of performing a wide variety of optimizations to generate efficient code for graphics engines. Sponge alleviates the problems associated with current GPU programming methods by providing portability across different generations of GPUs and CPUs, and a better abstraction of the hardware details, such as the memory hierarchy and threading model. Using streaming, we provide a write-once software paradigm and rely on the compiler to automatically create optimized CUDA code for a wide variety of GPU targets. Sponge's compiler optimizations improve the performance of the baseline CUDA implementations by an average of 3.2x.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Design, Languages, Performance

Keywords Streaming, Compiler, GPU, Optimization, Portability

1. Introduction

Support for parallelism in hardware has sharply grown in recent years as a response to performance and power demands of both emerging and traditional high performance application domains. Among the multitude of vastly different solutions offered by hardware companies, graphics processing units (GPUs) have been shown to provide significant performance, power efficiency and cost benefits for general purpose computing in highly parallel computing domains. Recently, heterogeneous systems that combine traditional processors with powerful GPUs have become standard in all systems ranging from servers to cell phones. GPUs achieve their high performance and efficiency by providing a massively parallel architecture with hundreds of in-order cores while exposing parallelism mechanisms and the memory hierarchy to the programmer. Recent works have shown that in the optimistic case, speedups of 100-300x [21] and in the pessimistic case, speedups of 2.5x [17] have been achieved between the most recent versions of GPUs compared to the latest processors. Maximizing the utilization of the GPU in heterogeneous systems will be key to achieving high performance and efficiency.

While GPUs provide an inexpensive, highly parallel system for accelerating parallel workloads, the programming complexity posed to application developers is a significant challenge. Developing applications to utilize the massive compute power and memory bandwidth requires a thorough understanding of the algorithm and details of the underlying architecture. Graphics chip manufacturers, such as NVIDIA, have tried to alleviate the complexity problem by introducing user-friendly programming models, such as CUDA [19]. Although CUDA and other similar programming models abstract the underlying GPU architecture by providing a unified processor model, managing the amount of on-chip memory used per thread, the total number of threads per multiprocessor, and the pattern of off-chip memory accesses are examples of problems that developers still need to manage in order to maximize GPU utilization [23]. Often the programmer must perform a tedious cycle of performance tuning to extract the desired performance.

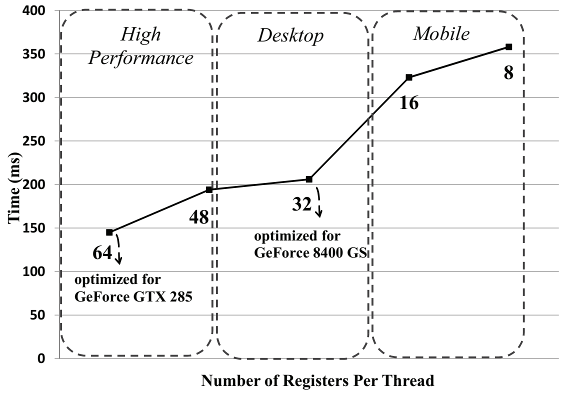


Figure 1: *This graph shows the runtime of a kernel optimized for architectures with different number of registers on a GeForce GTX 285 which has the most number of registers. The kernel used in this graph is organized in 128 blocks each with 256 threads.*

Another problem of developing applications in CUDA is the lack of efficient portability between different generations of GPUs and also between the host processors and GPUs in the system. Different NVIDIA GPUs vary in several key micro-architectural parameters such as number of registers, maximum number of active threads, and the size of global memory. These parameters will vary even more when newer high performance cards, such as NVIDIA's Fermi [20], and future resource-constrained mobile GPUs with less resources are released. These differences in hardware lead to a different set of optimization choices for each GPU. As a result, optimization decisions for one generation of GPUs are likely to be poor choices for another generation. An example of this is shown in Figure 1. This figure shows a CUDA kernel that requires 78 registers per thread, running with 128 blocks of 256 threads per block on an NVIDIA GeForce GTX 285. This graph shows how the runtime (lower is better) would change if the benchmark was optimized for GPU architectures with less than 16K registers available on each streaming multiprocessor of the GTX 285. For example, if this kernel is compiled for GeForce 8400 GS, it will use 32 registers per thread since there are 8K registers available for the 256 threads in each block on that architecture. Data elements that do not fit in the smaller register file will be spilled to the slower parts of the memory hierarchy causing performance degradation. In short, CUDA code must be separately customized for each target GPU as the choice of optimizations for peak performance is typically sensitive to the hardware configuration.

One solution to the GPU programming complexity is to adopt a higher level programming abstraction similar to the stream programming model. The streaming model provides an extensive set of compiler optimizations for mapping and scheduling applications to various homogeneous and heterogeneous architectures ([5, 6, 14]). The retargetability of streaming languages, such as StreamIt [26], has made them an excellent choice for parallel system programmers in shared/distributed memory and tiled architectures. Streaming language retargetability and performance benefits on heterogeneous systems are mainly a result of having well-encapsulated constructs that expose parallelism and communication without depending on the topology or granularity of the underlying architecture.

GPUs are important drivers for current and future heterogeneous systems, therefore extending the applicability of streaming languages to GPUs is advantageous for several reasons. First, streaming, which expresses programs at a higher level than CUDA, enables optimizing and porting to different generations of GPUs and between different topologies of CPUs and GPUs. Second, exposed communication in streaming programs help the compiler to efficiently map data transfers onto different memory hierarchies. Finally, streaming applications can be tailored for any number of cores and devices by performing graph restructurings such as horizontal or vertical fusion or fission of actors.

In this work, we introduce *Sponge*, a streaming compiler for the StreamIt language that is capable of automatically producing customized CUDA code for a wide range of GPUs. Sponge consists of stream graph optimizations to optimize the organization of the computation graph and an efficient CUDA code generator to express the parallelism for the target GPU. Producing efficient CUDA code is a multi-variable optimization problem and can be difficult for software programmers due to the unconventional organization and the interaction of computing resources of GPUs. Sponge is equipped with a set of optimizations to handle the memory hierarchy and also to efficiently utilize the processing units.

The Stream-to-CUDA compilation in Sponge consists of four steps. First, Sponge performs graph reorganization and modification on the stream graph and also classifies actors based on their memory traffic. The classification information is used throughout all the phases of the compilation. Second, memory layout optimizations are performed. These optimizations are designed to enable efficient utilization of the memory bandwidth. In this phase, Sponge decides if actors should use the faster but smaller on-chip memories or the slower but larger off-chip memory on the GPU. Also, techniques such as *helper threads* and *bank conflict resolution* in the context of StreamIt are introduced to increase the efficiency of memory accesses. The third compilation phase deals with actor size granularity of each thread. In this step, based on the classification information from step one, Sponge tries to create larger threads by fusing producer/consumer actors in order to reduce communication and kernel call overheads. Finally, software prefetching and loop unrolling are used to exploit unused registers to decrease loop control code overhead and increase memory bandwidth utilization.

In summary, this paper makes the following contributions:

- Extending applicability and portability of synchronous dataflow languages, specifically StreamIt, to GPUs.
- Streaming-specific optimizations for CUDA and generic CUDA optimizations for streaming applications.
- Discussion of the limitations of StreamIt as a GPU programming language.

The rest of the paper is organized as follows. In Section 2, the stream programming model, the input language (StreamIt), and the CUDA programming model are discussed. Portable stream compilation in Sponge and its optimizations are explained in Section 3. Experiments are shown in Section 4. A comparison between two hand-optimized CUDA benchmarks and their StreamIt implementation is done in Section 5. Related works are discussed in Section 6. Finally, Section 7 contains the conclusion.

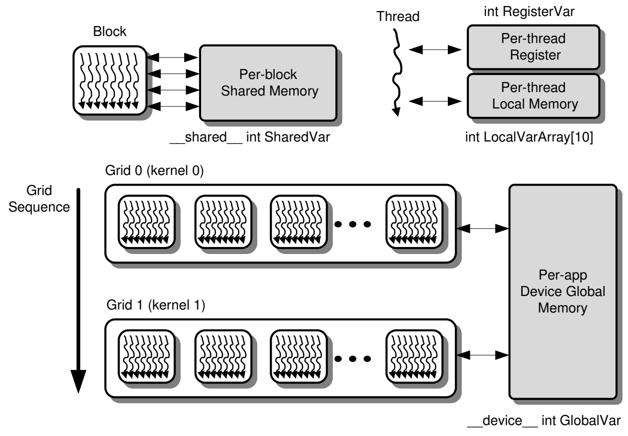


Figure 2: *CUDA/GPU Execution Model*

2. Background

In this section, the CUDA programming model, GPU architecture, and stream programming model are explained.

2.1 CUDA and GPUs

The CUDA programming model is a multi-threaded SIMD model that enables implementation of general purpose programs on heterogeneous GPU/CPU systems. There are two different device types in CUDA: the Host processor and the GPU. A CUDA program consists of a host code segment that contains the sequential sections of the program, which is run on the CPU, and a parallel code segment which is launched from the host onto one or more GPU devices. Data-level parallelism (DLP) and thread-level parallelism (TLP) are handled differently in these systems. DLP is converted into TLP and executed onto the GPU devices, while TLP is handled by executing multiple kernels on different GPU devices launched by the host processor. The threading and memory abstraction of the CUDA model is shown in Figure 2.

The threading abstraction in CUDA consists of three levels of hierarchy. The basic block of work is a single *thread*. A group of threads executing the same code are combined together to form a *thread block* or simply a *block*. Together, these thread blocks combine to form the parallel segments called *grids* where each grid is scheduled onto a GPU at a time. Threads within a thread block are synchronized together through a barrier operation ($\_syncthreads()$). However, there is no explicit software or hardware support for synchronization across thread blocks. Synchronization between thread blocks is performed through the global memory of the GPU, and the barriers needed for synchronization are handled by the host processor. Thread blocks communicate by executing separate kernels on the GPU.

The memory abstraction in CUDA consists of multiple levels of hierarchy. The lowest level of memory is *registers*, which are on-chip memories private to a single thread. The next level of memory

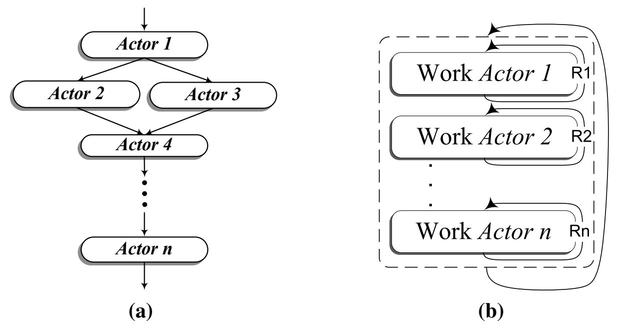


Figure 3: *This figure shows an example stream graph and also the intermediate code template for executing a steady state schedule. R_i is the repetition number for actor i.*

is *shared memory*, which is an on-chip memory shared only by threads within the same thread block. Access latency to both the registers and shared memory is extremely low. The next level of memory is *local memory*, which is an off-chip memory private to a single thread. Local memory is mainly used as spill memory for local arrays. Mapping arrays to shared memory instead of spilling to local memory can provide much better performance. Finally, the last level of memory is *global memory*, which is an off-chip memory that is accessible to all threads in the grid. This memory is used primarily to stream data in and out of the GPU from the host processor. The latency for off-chip memory is 100-150x more than that for on-chip memories. Two other memory levels exist on-chip called the *texture memory* and *constant memory*. Texture memory is accessible through special built-in texture functions and constant memory is accessible to all threads in the grid.

The CUDA programming model is an abstraction layer to access GPUs. NVIDIA GPUs use a single instruction multiple thread (SIMT) model of execution where multiple thread blocks are mapped to streaming multiprocessors (SM). Each SM contains a number of processing elements called Streaming Processors (SP). A thread executes on a single SP. Threads in a block are executed in smaller execution groups of threads called *warps*. All threads in a warp share one program counter and execute the same instructions. If conditional branches within a warp take different paths, called *control path divergence*, the warp will execute each branch path serially, stalling the other paths until all the paths are complete. Such control path divergences severely degrade the performance.

Because off-chip global memory access is very slow, GPUs support *coalesced memory accesses*. Coalescing memory accesses allows one bulk memory request from multiple threads in a half-warp to be sent to global memory instead of multiple separate requests. In order to coalesce memory accesses, three general restrictions apply: each thread in a half-warp must access successive addresses in order of the thread number, the memory accesses can only be 32, 64, or 128-bit, and all the addresses must be aligned to either 64, 128 or 256-byte boundaries. Effective memory bandwidth is an order of magnitude lower using non-coalesced memory accesses which further signifies the importance of memory coalescing for achieving high performance.

In modern GPUs, such as NVIDIA GTX 285, there are 30 SMs each with 8 SPs. Each SM processes warp sizes of 32 threads. The memory sizes for this GPU are: 16K of registers per SM, 16KB divided into 16 banks of shared memory per SM, and 2GB of global memory shared across all threads in the GPU.

2.2 Stream Programming Model

With the ubiquity of multi-core systems, the stream programming paradigm has become increasingly important. Exposed communication and an abundance of parallelism are the key features making streaming a flexible and architecture-independent solution for parallel programming. In this paper, we focus on stream programming models that are based on synchronous data flow (SDF) models [15]. In SDF, computation is performed by actors, which are autonomous and isolated computational units. Actors communicate through data-flow buffers (i.e. tapes), often realized as FIFOs. SDF, and its many variations, expose the input and output processing rates of actors, and in turn this affords many optimization opportunities that can lead to efficient schedules (e.g., allocation of actors to cores, and tapes to local memories).

For our purpose, we assume all computation that is performed in an actor is largely embodied in a *work* method. The *work* method runs repeatedly as long as the actor has data to consume on its input port. The amount of data that the work method consumes is called the *pop* rate. Similarly, the amount of data each work invocation produces is called the *push* rate. Some streaming languages (e.g., StreamIt [26]) provide a non-destructive read which does not alter the state of the input buffer. The amount of data that is read in this manner is specified by the *peek* rate. An actor can also have an *init* method that is executed only once for the purpose of initializing the actor before the execution of program starts.

We distinguish between stateful and stateless actors. A stateful actor modifies its local state and maintains a persistent history of its execution. Unlike a stateful actor, which restricts opportunities for parallelism, a stateless actor is data-parallel in that every invocation of the work method does not depend on or mutate the actor's state. The semantics of stateless actors thus allow us to replicate a stateless actor. This opportunity is quite fruitful in scaling the amount of parallelism that an application can exploit, as shown in [5, 6].

We use the StreamIt programming language to implement streaming programs. StreamIt is an architecture-independent streaming language based on SDF. The language allows a programmer to algorithmically describe the computational graph. In StreamIt, actors are known as filters. Filters can be organized hierarchically into *pipelines* (i.e., sequential composition), *split-joins* (i.e., parallel composition), and *feedback loops* (i.e., cyclic composition). StreamIt is a convenient language for describing streaming algorithms, and its accompanying static compilation technology makes it suitable for our work.

A crucial consideration in StreamIt programs is to create a steady state schedule which involves rate-matching of the stream graph. Rate-matching guarantees that, in the steady state, the number of data elements that is produced by an actor is equal to the number of data elements its successors will consume. Rate-matching assigns a static repetition number to each actor. In the implementation of a StreamIt schedule, an actor is enclosed by a *for-loop* that iterates as many times as its repetition number. The steady state schedule is a sequence of appearances of these *for-loops* enclosed in an outer-loop whose main job is to repeat the steady schedule. The template code in Figure 3b shows the inter-

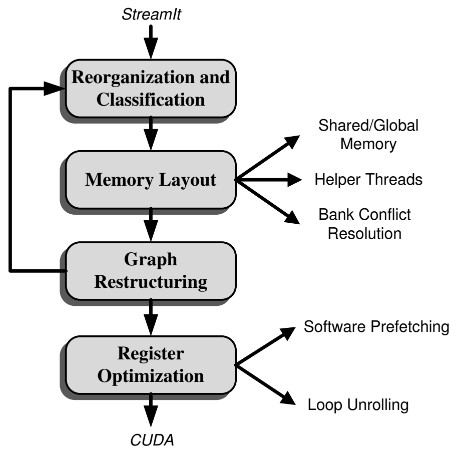


Figure 4: *Compilation flow in Sponge.*

$$ActiveWarpsPerSM = \frac{ThreadsPerBlock \times ActiveBlocksPerSM}{THREADS\_PER\_WARP} \quad (1)$$

$$Iterations = \frac{InputBufferSize}{Pop \times ThreadsPerBlock \times Blocks} \quad (2)$$

$$ThreadsPerBlock_{LoT} = \frac{SHARED\_MEMORY\_SIZE}{(Pop + Push)} \quad (3)$$

$$ExecCycles_{LoT} = \frac{CompInsts \times COMP\_INST\_ISSUE\_DELAY}{NUMBER\_SM} \times \frac{ThreadsPerBlock_{LoT} \times Iterations}{ActiveWarpsPerSM} \quad (4)$$

$$ThreadsPerBlock_{HiT} = MAX\_THREAD\_PER\_BLOCK \quad (5)$$

$$\begin{aligned} MemCycles &= (UncoalMemInsts + CoalMemInsts/COAL\_FACTOR) \\ &\quad \times MEMORY\_DELAY + MEM\_INST\_ISSUE\_DELAY \end{aligned} \quad (6)$$

$$ExecCycles_{HiT} = \frac{MemCycles}{NUMBER\_SM} \times \frac{ThreadsPerBlock_{HiT} \times Iterations}{ActiveWarpsPerSM} \quad (7)$$

Name	Description	Name	Description
$SHARED\_MEMORY\_SIZE$	Size of shared memory on GPU	$pop, push$	push and pop rate of an actor
$THREADS\_PER\_WARP$	Number of threads in each warp	$InputBufferSize$	Size of input buffer for an actor
$NUMBER\_SMs$	Number of streaming processor on GPU	$ThreadsPerBlock$	Number of threads in one block
$MAX\_THREAD\_PER\_BLOCK$	Max number of threads allowed per block	$Blocks$	Number of blocks on the GPU
$MEMORY\_DELAY$	Number of cycles to access global memory	$Iterations$	Number of iterations to run an actor on the GPU
$COAL\_FACTOR$	Max number of memory accesses that can be coalesced	$ActiveBlocksPerSM$	Blocks active on one SM
$MEM\_INST\_ISSUE\_DELAY$	Number of cycles to issue a memory instruction	$(Un)CoalMemInsts$	(Un)Coalesced instructions in one actor
$COMP\_INST\_ISSUE\_DELAY$	Number of cycles to issue a compute instruction		

Figure 5: *In this Figure, equations for calculating execution cycles of both HiT and LoT actors are shown. Equations 1 and 2 can be used for both HiT and LoT actors. The table summarizes what each variable means.*

mediate code for the steady state schedule of the streaming graph shown in Figure 3a.

3. Portable Stream Compilation

Sponge takes StreamIt programs as its input and generates GPU-specific CUDA code. Each actor in the StreamIt graph is converted to a CUDA kernel running with some number of threads and blocks. By performing portable stream compilation, Sponge decides how many threads and blocks to assign to the CUDA kernel generated for each actor. The input buffer size of the first actor, A_i, in the graph determines how many times that actor has to run (R_i). As a result, R_i is changed to R_i divided by the multiplication of number of threads and blocks assigned to the actor. We call the result *number of iterations* for actor A_i.

Portable stream compilation in *Sponge* consists of four main steps as shown in Figure 4. In the first phase, Sponge reads a StreamIt program and performs *Actor Reorganization and Classification* in which simple graph reorganization is done and actors are classified into two categories: *High-Traffic (HiT) and Low-Traffic (LoT)*. The classification information is used throughout all the phases of the compilation flow. The second phase deals with the *Memory Layout and Optimization* of each actor. This step decides if an actor uses shared or global memory, eliminates shared memory bank conflicts and also improves memory performance by introducing *Helper Threads* to better utilize the unused processors and bring the data needed by an actor into shared memory faster. This compilation step is crucial to achieving better performance since memory bandwidth can be a limiting factor on GPUs. The third phase performs *Graph Restructuring* by changing the granularity of the kernels and vertically fusing actors based on classification results. After graph restructuring, the compiler reiterates from the beginning of the compilation flow, treating the post-fused stream graph as the input until no more graph restructuring is possible. Finally, *Register Optimization* tries to utilize unused registers on each SM by employing software prefetching and also by unrolling *for loop*s in each kernel.

3.1 Actor Reorganization and Classification

As mentioned in Section 2.1, GPUs are built for data-level parallelism and are not suitable for task-level parallelism and global synchronization. Therefore, *splitter-joiner* structures will not perform well on the GPU since each joiner introduces a synchronization point. First, Sponge collapses *splitter-joiner*s to one actor in cases that the actors between the *splitter* and *joiner* are stateless and equivalent. This will remove the *splitter* and *joiner* actors and replace the structure with a single actor. In cases where it is not possible to collapse a *splitter-joiner* structure to one actor, Sponge treats the *splitter* and *joiner* as special actors with more than one input and output. Based on the type and weights of the *splitter* and *joiner* actors, Sponge decides to allocate their input and output buffers in shared memory or global memory.

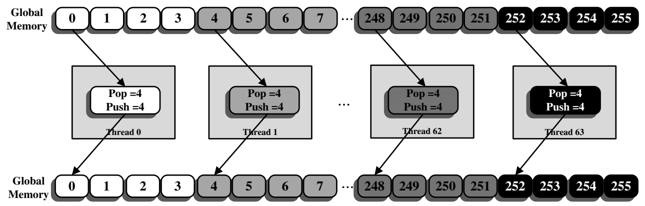
Sponge excludes stateful actors from being executed on the GPU and runs them on the host CPU. This is because only one instance of a stateful actor can be active and data-parallelism is not applicable to these actors. Host to GPU and GPU to host transfers are inserted before and after stateful actors, if necessary.

Next, Sponge classifies actors assigned to the GPU as either High-Traffic (HiT) or Low-Traffic (LoT). HiT actors have a large number of memory accesses. These actors perform better on a GPU if their buffers are mapped to global memory rather than shared memory because mapping the buffers to shared memory will result in having too few threads and under-utilizing the processors and the available memory bandwidth. LoT actors, on the other hand, are mostly computation dominated and if mapped to shared memory will have a reasonable number of threads to utilize the GPU.

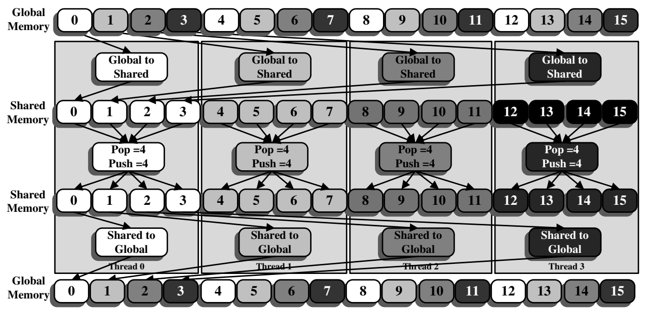
In order to determine if an actor is a LoT or HiT, Sponge estimates execution cycles of an actor for both global memory (HiT) and shared memory (LoT) mappings, based on Equations 1-7 in Figure 5. For each actor, Sponge treats that actor as both HiT and LoT and calculates the corresponding execution cycles ($ExecCycles_{LoT}$, $ExecCycles_{HiT}$). The two numbers show if that actor is suitable to be treated as a LoT or HiT actor. If $ExecCycles_{HiT}$ is smaller than $ExecCycles_{LoT}$ for an actor, that actor will perform better if its buffer is mapped to global memory. Otherwise, it will be classified as a LoT actor for which both shared and global memory will be used to help with coalescing of data accesses.

In the equations for LoT actors, number of threads per block ($ThreadsPerBlock$) is determined by the size of shared memory($SHARED\_MEMORY\_SIZE$ [1]) and the number of pushes and pops. Threads per block defines the number of active warps per SM ($ActiveWarpsPerSM$) and the number of iterations based on Equations 1 and 2. Finally, the execution cycle of a LoT actor is estimated depending on the number of compute instructions and the distribution of threads in the GPU (Equation 4).

[1] Variables with all capital characters show GPU-specific parameters



(a)



(b)

Figure 6: *This figure shows how HiT and LoT threads access their buffers. Part (a) illustrates the memory access pattern for a sample HiT actor with four pops and four pushes. Part (b) shows the access pattern for a LoT actor.*

Execution time of HiT actors is calculated based on their memory access time because these actors are mapped to global memory and have a large number of global memory reads and writes. Equations 5-7 show how execution time estimation is done based on the number of coalesced and uncoalesced memory accesses. Unlike shared memory, the size of global memory does not limit the number of threads. Therefore, the number of threads per block for HiT actors can be equal to the maximum number of threads allowed in each block ($MAX\_THREAD\_PER\_BLOCK$).

In this section, memory layout and optimization techniques used in Sponge are discussed. First, the way shared memory is utilized for LoT actors is explained. Second, helper threads, a technique that Sponge uses to reduce global memory access time of actors, is discussed. Finally, shared bank conflict resolution in Sponge is explained.

3.1.1 Shared/Global Memory

To deal with high-latency memory access issues, Sponge uses the classification information calculated in the previous phase and tries to alleviate the problem by coalescing the buffer accesses or overlapping a large number of uncoalesced buffer accesses to amortize the cost. As discussed earlier, HiT actors will be mapped to global memory and LoT actors will use the shared memory. The kernel generated for these actors will have a large number of threads, each accessing its own buffer sequentially in global memory. The memory accesses will not be coalesced because the accesses of consecutive threads are not consecutive in the memory. Since the number of threads is large, the overhead of memory accesses will be hidden by the execution of many threads. Figure 6a illustrates how a HiT actor with four pops and four pushes accesses global memory. Since the addresses generated by the first pop operations of the threads are not consecutive in the memory, they are not coalesced.

LoT actors, unlike HiT actors, have a higher compute to memory ratio. Therefore, a LoT actor can use shared memory and have a large number of threads. As shown in Figure 6b, threads of a LoT kernel in a block can use coalesced memory accesses to copy their input (output) buffer to (from) shared memory from (to) global memory. To do so, the threads of a block work as a group and bring parts of data that belong to other threads as well as part of their own data. In this way, consecutive threads' accesses to shared memory will be to consecutive locations and will get coalesced. Since all of the data is in shared memory, all threads in a block will have access to it. Figure 7 shows how the CUDA code needs to be changed to utilize shared memory in LoT actors. In the baseline form (Figure 7a) the input and output buffers are allocated in global memory and the work function directly accesses global memory. If shared memory is used, then two *for loop*s are added before and after the work function to copy the data in and out of shared memory, as shown in Figure 7b. The addresses for the memory reads and writes in these *for loop*s are set based on the $ThreadID$, and the number of pushes and pops. Before and after the two new *for loop*s, L_1, L_2, barriers (*synchthreads*) are necessary because, as mentioned earlier, each thread does not fetch all of its own data and has to wait for other threads in the block to finish their data-fetch phase.

3.2 Memory Layout and Optimization

Memory hierarchy in GPUs is significantly different from both conventional shared memory and distributed memory systems. As mentioned in Section 2.1, efficient use of global memory, shared memory and registers on GPUs is crucial to obtain high performance. Coalescing accesses to global memory can greatly reduce memory access overheads, but it will not be possible without careful memory layout. Utilizing shared memory, which is significantly faster than global memory, is also very beneficial. Due to its limited size, shared memory can restrict the number of threads and degrade the performance. In this section, techniques used for memory layout and optimization in Sponge are discussed.

3.2.1 Helper Threads

The first optimization of this phase is to use $helper$ threads to fetch data for the $worker$ threads. In cases where there are not enough threads to efficiently utilize all the SMs for LoT kernels or a HiT actor has a fair number of threads when it is treated as a LoT actor (mapped to shared memory), Sponge uses helper threads to reduce the buffer (i.e. memory) accesses of each thread (push and pop rate). Each helper thread aids some worker threads to bring their data to shared memory in a coalesced way.

Figure 7c shows how the CUDA code is modified. Based on the thread IDs, Sponge generates the helper and worker threads. Helper threads are in charge of handling the data accesses and worker threads are in charge of the computation. In order to avoid control flow divergence, the thread assignment is performed such that the helper and worker threads form complete warps. If the number of worker threads are less than the warp size, then the helper threads are placed in the first set of warps and the worker threads form the last warp. This is done by predicating out the work function for the helper threads and the memory access *for loop*s for the worker functions. The *if* statement in Figure 7c do this based on $threadID$. This technique works because control flow divergence negatively affects the performance within one warp but not across warps.

Sponge estimates number of instructions that helper threads will add to each thread and also takes into account the parallelism between the helper and worker threads to calculate how beneficial helper thread optimization will be for both LoT and HiT threads. As illustrated in Figure 7c, Sponge counts the time it takes to run L_1, L_2, and L_3 sections and estimates the total execution time based on the equations in Figure 5. If the total execution time using helper threads is reduced, Sponge generates CUDA code using them.

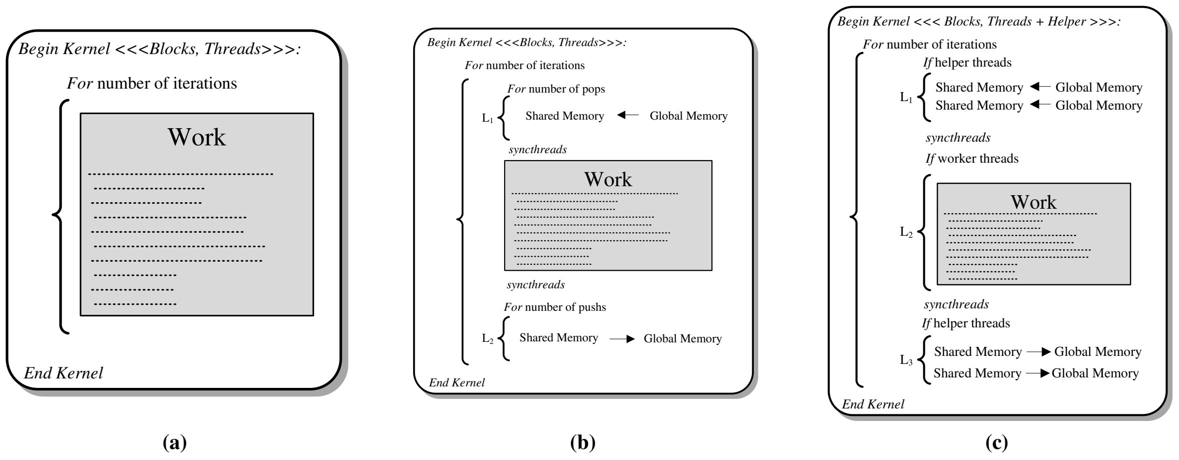


Figure 7: *Part (a) shows the baseline translation for a HiT actor. How shared memory is used in a LoT actor is illustrated in part (b). In part (c) the way Sponge generates CUDA code to divide threads as helpers and workers is shown.*

3.2.2 Bank Conflict Resolution

Shared memory bank conflict is another source of bottleneck in GPU systems. For example, whenever threads of a kernel access their input buffer in shared memory with:

$$data \quad = \quad buffer[baseAddress + s * threadId];$$

threads $threadId$ and $threadId + n$ access the same bank whenever n is a multiple of m/d (m is the number of memory banks) where d is the greatest common divisor of m and s. As a consequence, there will be no bank conflicts only if half the warp size is less than or equal to m/d. For current NVIDIA devices, this translates to no bank conflict only if d is equal to 1, or in other words, only if s is odd since m is a power of two (16 for GTX 285 [19]). In the StreamIt code, s is the number of pops. To make s odd, if the number of pops is even, Sponge artificially changes the pop rate of an actor by incrementing the pops by one. In this way, an actor with $2k$ pops will use $2k+1$ entries in the memory and the buffers get shifted in the memory. Removing bank conflicts greatly improves the performance of some of the benchmarks. The same technique can be applied for pushes.

3.3 Graph Restructuring

In this part, Sponge vertically fuses some actors to improve performance by increasing coalesced memory accesses, removing kernel call overhead, and also increasing instruction overlap. Fusion is not beneficial in all cases because it can increase the memory traffic (push + pop) of a pair of LoT actors and reduce the number of threads (Equation 3). For HiT actors, fusion may increase the memory traffic as a result of register spilling.

The main benefit of fusing HiT actors is replacing uncoalesced memory accesses at the end of the first actor and at the beginning of the second actor with coalesced accesses. The memory accesses become coalesced because the two actors within the fused actor are rate matched. Therefore, the first actor can write to the internal buffer using coalesced memory writes and the second actor can read the same data with coalesced memory reads. Figure 8 illustrates how fusion can lead to coalescing of memory accesses in a simple GPU that has warp size of four and can coalesce two memory accesses into one. In this figure, the memory accesses between actors A (producer with 2 pushes running with 8 threads) and B (consumer with 8 pops running with 2 threads) are shown. $W_{i,j}$ is jth push by the ith thread of A, and $R_{k,m}$ is the mth pop of the kth thread of B. Figure 8a shows how writes and reads are performed between these actors in the case of no fusion. In this case each thread serially writes and reads from global memory which results in all uncoalesced accesses (marked by U). If the buffer allocation for A is changed such that its memory accesses can be coalesced (marked by C), as shown in Figure 8b, the accesses of threads running B will still be uncoalesced. Figure 8c shows the accesses to the internal buffer between A and B after fusion is performed. The new actor, $(4A)B$, runs with two threads. Since there are 8 pushes and pops between $4A$ and B all the accesses will be coalesced, as shown in Figure 8c.

For the LoT actors, global memory accesses are already coalesced with the help of shared memory. These accesses happen in two *for loop*s before and after the work function. Similar to the HiT case, the accesses between the two LoT actors become coalesced. Therefore, the resulting LoT actor does not need to use shared memory anymore. This will result in elimination of a large number of complex address calculations and *for loop* control instructions.

Sponge uses its cost estimation equations to decide if fusing a pair of actors is beneficial or not. For a candidate pair, Sponge calculates the number of cycles for both cases where the resulting actor is HiT or LoT. If in either case the execution time is less than the sum of the original actors' execution times, fusion is performed.

3.4 Register Optimization

Registers on GPUs are a precious resource. Efficiently using the registers can greatly improve performance. In this section, two optimizations that Sponge performs to increase register utilization are discussed.

3.4.1 Software Prefetch

To better tolerate long memory access latency, the CUDA threading model allows some warps to make progress while others wait for their memory access results. This mechanism is not effective in some cases where all threads are waiting for their memory access results. This case happens if all threads have very few independent instructions between memory access instructions and the use of the accessed data. Prefetching is a technique that some CUDA programs use to overlap fetching data from global memory for iteration $i+1$ of an actor with compute instructions in iteration i by utilizing the available registers.

Figure 9a shows how software prefetching can be done for LoT actors. Before the main *for loop*, the first batch of data (for iteration 1) is loaded into registers (L_1). Once L_2 has started, the data is moved from registers into shared memory. At this point, threads have to wait for the shared memory transfers to finish before they

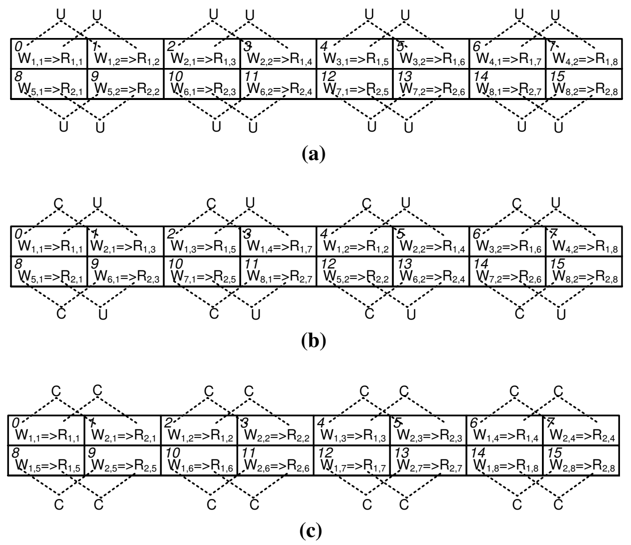

(a)
0 W1,1=>R1,1 | 1 W1,2=>R1,2 | 2 W2,1=>R1,3 | 3 W2,2=>R1,4 | 4 W3,1=>R1,5 | 5 W3,2=>R1,6 | 6 W4,1=>R1,7 | 7 W4,2=>R1,8
8 W5,1=>R2,1 | 9 W5,2=>R2,2 | 10 W6,1=>R2,3 | 11 W6,2=>R2,4 | 12 W7,1=>R2,5 | 13 W7,2=>R2,6 | 14 W8,1=>R2,7 | 15 W8,2=>R2,8
U U U U U U U U
(b)
0 W1,1=>R1,1 | 1 W2,1=>R1,3 | 2 W1,3=>R1,5 | 3 W1,4=>R1,7 | 4 W1,2=>R1,2 | 5 W2,2=>R1,4 | 6 W3,2=>R1,6 | 7 W4,2=>R1,8
8 W5,1=>R2,1 | 9 W6,1=>R2,3 | 10 W7,1=>R2,5 | 11 W8,1=>R2,7 | 12 W5,2=>R2,2 | 13 W6,2=>R2,4 | 14 W7,2=>R2,6 | 15 W8,2=>R2,8
C U C U C U C U
(c)
0 W1,1=>R1,1 | 1 W2,1=>R2,1 | 2 W1,2=>R1,2 | 3 W2,2=>R2,2 | 4 W1,3=>R1,3 | 5 W2,3=>R2,3 | 6 W1,4=>R1,4 | 7 W2,4=>R2,4
8 W1,5=>R1,5 | 9 W2,5=>R2,5 | 10 W1,6=>R1,6 | 11 W2,6=>R2,6 | 12 W1,7=>R1,7 | 13 W2,7=>R2,7 | 14 W1,8=>R1,8 | 15 W2,8=>R2,8
C C C C C C C C

Figure 8: *This figure shows the memory accesses between actors A with 2 pushes and 8 threads and B with 8 pops and 2 threads. $W_{i,j}(R_{i,j})$ shows jth memory write (read) performed by ith thread running actor A (B). U and C denote uncoalesced and coalesced. Part (a) shows the accesses in the base case. Part (b) illustrates the same accesses when the buffer for A is allocated such that its writes are coalesced. Part (c) shows coalesced accesses between these two actors when they are fused as $(4A)B$ and executed with two threads. The number on the top left corner of each box shows the memory address of that location.*

can progress because that data is needed for computation after this point. After all threads are done moving the data to shared memory, they pass the barrier synchronization point and begin to load the next batch of data into registers. The key is that the work function does not need the data from these memory accesses and overlapping of compute and memory accesses can happen. Loop L_4 has to be wrapped in an *if* statement because the last iteration of the kernel does not need to prefetch any data. This *if* statement does not introduce branch divergence since all the threads take the same path at this point.

One possible downside of this technique is that using additional registers for prefetching can reduce the number of blocks that can run on an SM. However, prefetching is beneficial if it significantly reduces the amount of time each thread waits for global memory accesses. Since different classes of NVIDIA GPUs are equipped with different number of registers, Sponge tunes this optimization for each GPU target. If performing prefetching for the whole buffer introduces register spill or reduce the number of concurrent blocks, Sponge tunes the prefetching optimization by applying it to only a fraction of the input buffer.

3.4.2 Loop Unrolling

Instruction processing bandwidth on the processing cores of current CUDA graphics engines can negatively affect the performance of an actor. Address calculation and loop control instructions can become important if an actor has small number of computation instructions. In other words, these type of instructions introduce overhead and prevent a kernel from utilizing the peak performance of a GPU. Loop unrolling is one way to reduce the overhead. This optimization can also increase the register utilization by unrolling loops that use registers. The degree of unrolling depends on the number of registers the kernel uses and also the number of registers that are available on the GPU. Since different classes of GPUs are equipped with different number of registers per SM, blindly applying unrolling to the *for-loop*s in a kernel may worsen the performance.

An example of the unrolling is shown in Figure 9b. There are five potential *for loop*s in a typical LoT actor generated by Sponge, as shown in Figure 9a, 2 for transferring data to and from global memory, 2 for prefetching and 1 for the work function. Depending on the number of registers available on the target GPU and the instruction mix of the kernel, Sponge decides to perform loop unrolling on the *for loop*s.

Figure 9b shows how the unrolling is applied to all five *for loop*s to both remove the *for loop* overheads and also increase the register utilization. In this example, unrolling factor of two is applied to the *work* function. As shown in Figure 9a, loop L_1 is unrolled to U_1. Because the work function is unrolled two times, all the corresponding *for loop*s now appear twice except L_4. Replicating L_4 two times will result in having two *if* statements. To remove the conditional branch instruction overhead, these two replicated *for loop*s are merged into U_6.

3.5 A Stream Compilation Example

In this section, a running example, as shown in Figure 10, is used to better illustrate how the optimizations affect the streaming graph. The base graph in Figure 10a has 12 unique actors two of which are in a *splitter-joiner* structure. Each box shows one actor in the program. Each edge in this graph indicates a tape implemented using FIFO queues. The text written inside each box shows how each actor interacts with its input and output tapes. All the actors are stateless except G. This actor as well as the source (A) and the sink (H) actors are mapped to the host processor.

In the classification phase, Sponge will remove the two *splitter*s and *joiner*s and replace all the copies of C and E with one of each. This is done because GPUs do not support task level parallelism and the *joiner* will introduce synchronization overhead. After this, actors are classified as HiT and LoT based on their memory traffic and computation instructions. LoT actors use shared memory but HiT actors operate on global memory. In the example, actors D and F are identified as HiT actors (shown with a darker color) and B, C and E as LoT actors. $[i, j]$ next to each GPU actor shows number of threads (i) and number of blocks (j) that will run that actor. For the LoT actors, the number of threads depends on the size of shared memory and the memory usage of the actor. For HiT actors, the number of threads is always equal to the maximum number of threads allowed per block because global memory is significantly larger than the actors' memory footprint.

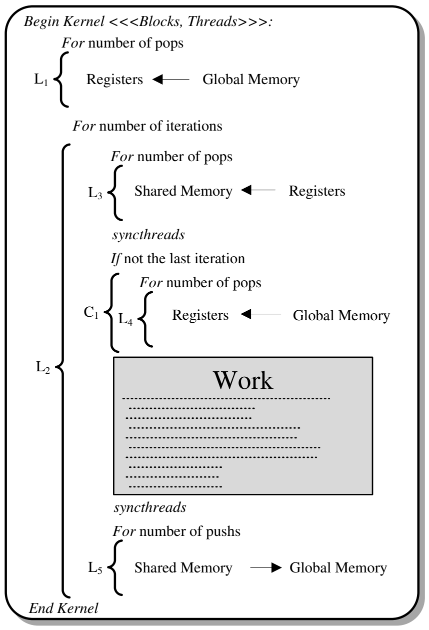
Next, helper threads are used to fetch data from global memory to shared memory more efficiently. After applying this, the number of threads for LoT actors will increase but HiT actors will be running with less threads because they have to use the shared memory. In the example, as shown in Figure 10c, except actor D, every actor benefits from using helper threads. If the number of threads for an actor is written as $w + h$, then w shows the number of worker threads and h shows number of helper threads assigned to that actor.

Finally, graph restructuring is performed on the graph and as a result several actors get fused together. Figure 10d shows the result of fusion and then re-applying classification and helper thread optimization. Actor B and C are fused together in a LoT actor and actors D, E, and F are classified as a HiT actor.

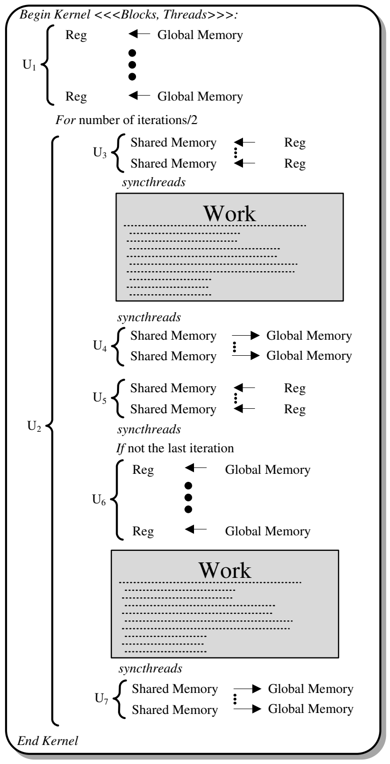
4. Experiments

In this section Sponge's optimization techniques are evaluated and compared with two alternative approaches:

1. GPU baseline: All stateless actors of the benchmarks are mapped to the GPU utilizing the maximum number of threads supported ($MAX\_ THREAD\_ PER\_ BLOCK$). In this technique, all of the actors are compiled as HiT actors. Stateful actors as well as source and sink actors are mapped to the host processor.
2. CPU baseline: All the actors are executed sequentially on the CPU.



(a)



Begin Kernel <<<Blocks, Threads>>>:
U1
Reg
Global Memory
For number of iterations/2
U3
Shared Memory
Reg
syncthreads
Work
U4
Global Memory
U5
U2
If not the last iteration
U6
U7
End Kernel

(b)

Figure 9: *Part (a) shows how prefetching is performed to improve the performance of a kernel. Part (b) depicts the result of unrolling on the kernel in part (a).*

4.1 Methodology

A set of benchmarks from the StreamIt suite [26] are used to evaluate Sponge. The benchmarks are compiled and evaluated on a system containing a 3GHz Intel Core 2 Extreme CPU with 6GB of RAM and a GeForce GTX 285 GPU with 2GB DDR3 global memory. Sponge compilation phases are implemented as a compiler backend to the StreamIt compiler. Sponge generates customized CUDA code which is compiled using NVIDIA nvcc 3.1 for execution on the GPU. GCC 4.1 is used to generate the x86 binary for execution on the host processor.

4.2 Techniques Performance

In this section, we try to compare the Sponge optimization techniques to the GPU baseline and highlight the effectiveness of each optimization. Figure 11a shows how Sponge-generated CUDA code performs and shows the performance gain of each optimization technique. On average, Sponge improves the performance by 3.2x compared to the GPU baseline.

The first optimization, shared/global memory, which divides actors into two categories LoT and HiT, is one of the most beneficial Sponge techniques. By using shared memory, Sponge is able to coalesce all the memory accesses in LoT actors, therefore performance of benchmarks containing LoT actors will significantly increase. As shown in the Figure 11a, *Matrix Multiply Block* benefits the most because this benchmark has several LoT actors. As a result, most of the actors in *Matrix Multiply Block* have coalesced memory accesses. In some benchmarks, such as *Histogram*, little benefit is seen using this optimization because most actors are HiT actors.

Prefetching and unrolling are two other optimizations illustrated in Figure 11a. These optimizations, collectively, contribute to 3.1% of the total average speedup. Prefetching technique is used only for LoT actors and is useful mostly in applications with many LoT actors such as *Merge sort* and *Bitonic*. Unrolling allows Sponge to utilize unused registers and reduce the number of instructions. This technique can increase the performance of LoT actors that use few registers. *DCT*, *Merge Sort*, *Radix*, and *Bitonic* have such actors and unrolling can increase their performance.

Another effective optimization in Sponge is employing helper threads. As described in the previous sections, helper threads can reduce the execution time of both LoT and HiT actors with two exceptions:

- LoT actors with many threads: In this case, it is not possible to run more threads to help the worker threads. Reducing the number of worker threads would decrease performance.
- HiT actors with few threads: To utilize helper threads, HiT actors would be converted into LoT actors, which have less threads because of the limited shared memory size. Though transferring data to shared memory improves memory performance, too few worker threads can become a bottleneck, underutilizing the SMs and decreasing the overall performance.

As shown in Figure 11a, helper thread optimization effectively increases the performance of *DCT*, *FFT*, *Matrix multiply* and *Merge sort*. For example *DCT* has multiple HiT actors with a large number of worker threads. In this case, coalescing data accesses using shared memory provides enough performance gain that running the actors with less threads will not result in slowdown. On average, helper threads contributes to 16% of the total average speedup compared to the GPU baseline.

Graph restructuring decreases the overhead of kernel launching and uncoalesced memory accesses. As discussed in Section 3.3, there are some cases where fusing two actors may result in degraded performance. Since two actors that are fused must execute together, the number of threads that the resulting actor can run will be less than the number of threads created by running each ac-

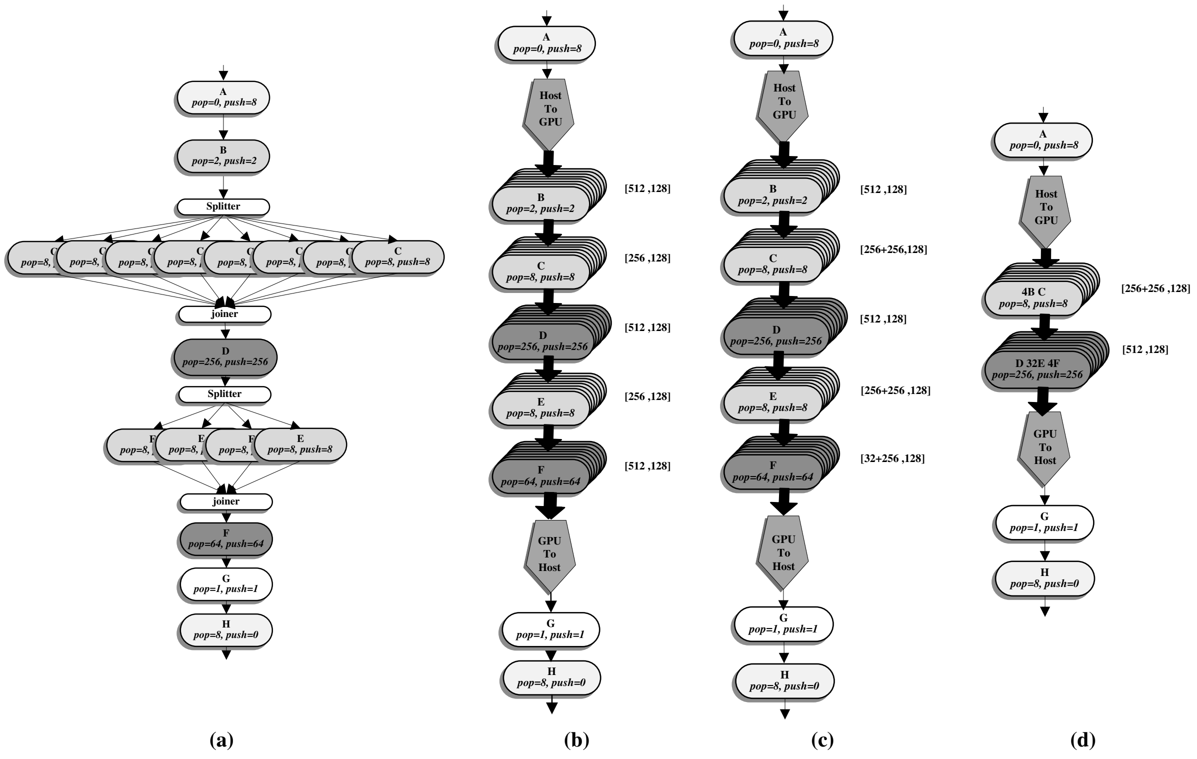
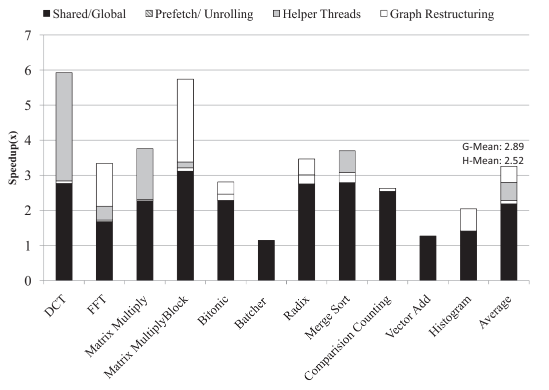
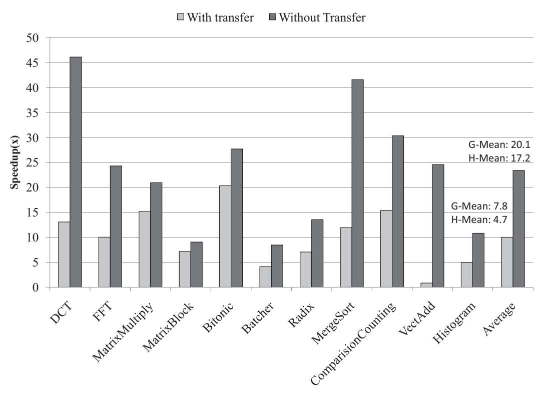


Figure 10: *Part (a) shows a stream graph with 12 unique actors. Part (b) is about how actor classification and graph reorganization affects this graph. In this part, shaded actor are HiT actors. Part (c) illustrates the result of the helper thread optimization. Part (d) depicts the same graph after applying graph restructuring.* $[i, j]$ *next to each GPU actor shows number of threads* (i) *and number of blocks* (j) *that will run that actor. If* i *is written as* $w + h$, w *is number of worker threads and* h *is the number of helper threads.*



(a) *Performance breakdown of Sponge optimizations in comparison to the baseline CUDA code, both running on the GPU.*



(b) *Speedup of Sponge optimized code in comparison to the host CPU with and without data transfer overhead.*

Figure 11: *Effectiveness of Sponge optimization techniques on StreamIt benchmarks.*

tor separately. Because the reduction of threads can decrease the performance, Sponge intelligently decides whether or not to use this optimization. Several benchmarks, such as *FFT*, have large pipelines of actors that are all fused together by Sponge. Graph restructuring provides a large portion of the speedup for these types of benchmarks. Since *Batcher* and *Vector Add* have only one actor, fusion cannot increase their performance. In *Merge Sort*, the opportunity for performing fusion is minimal because most of the actors in this benchmark are isolated from each other and do not form a pipeline.

4.3 Overall performance

Figure 11b presents the speedup of Sponge's generated CUDA applications against the CPU baseline, both with and without the data transfer time between the GPU and CPU. On average, Sponge achieves about 20x speedup compared to running each benchmark completely on the CPU. The only case that the CPU baseline outperforms Sponge is *Vector Add* including the data transfer overhead. In this special case, the memory to compute ratio in *Vector Add* is very high. Although the GPU can execute the *Vector Add* actor 10x faster than CPU, the overhead of transferring the data

GTX 285											
	DCT	FFT	MM	MM Block	Bitonic	Batcher	Radix	Merge Sort	Comp Count	Vect Add	Histogram
Shared	66.7	66.7	62.5	80	100	100	100	100	100	100	33.3
Prefetch	0	4.2	12.5	0	40.7	0	0	34.8	13	0	0
Unrolling	50	70.8	37.5	80	50.7	100	100	30.4	63.2	0	33.3
Helping Threads	50	16.7	25	20	0.7	0	0	52.1	0	0	0
Tesla C2050											
Shared	100	100	87.5	93.3	100	100	100	100	100	100	33.3
Prefetch	0	41.7	12.5	6.7	1.5	0	100	0	100	0	66.7
Unrolling	50	33.3	50	60	34.4	100	0	0	0	0	0
Helping Threads	50	0	25	6.7	0.7	0	0	0	0	0	33.3

Table 1: *This table shows how Sponge optimizes each benchmark differently for two GPU targets. For each benchmark and target, the percentage of actors that are optimized by each optimization is shown.*

between the host and GPU global memory decreases the overall performance.

4.4 Portability

Quantifying portability is inherently a hard problem. To show how Sponge solves the portability issue, we show how it optimizes each benchmark differently for two GPUs, Tesla C2050 and GeForce GTX 285. The C2050 is based on the newer NVIDIA architecture (Fermi) which has 48KB of shared memory, 32K registers and 420 streaming processors, providing Sponge with more resources to exploit. Table 1 shows how Sponge makes different decisions based on the target architecture. This table illustrates the percentage of actors in each benchmark optimized using various optimizations in Sponge. As shown in Table 1, Sponge is able to classify more actors as LoT actors and utilize the larger shared memory in C2050. The number of registers also affects how Sponge performs unrolling and prefetching for each target. In general, Sponge adopts its compilation strategy based on the characteristics of the GPU target without any source code modification or programmer involvement.

5. Case Study and Future Work

Sponge is designed to reduce the performance gap between automatically generated CUDA programs and hand-optimized ones. In this section two hand-optimized CUDA programs from the NVIDIA SDK are analyzed to highlight the reasons for performance differences between Sponge-generated and hand-optimized CUDA code.

5.1 Black-Scholes

The *Black-Scholes* algorithm is a differential equation that can predict how the value of an option changes. This equation reads five parameters from the input data and computes the price for an option call and an option put and writes these two values to the output. In the code generated by Sponge for GTX 285 GPU, there is only one kernel that pops five memory element from the input and calculates the output and pushes two results to the output buffer. This actor is classified as an LoT actor. Therefore, Sponge uses shared memory to coalesce all the buffer accesses in that actor. In the hand-optimized code, only one kernel is launched as well, but each parameter is placed in a different array. The kernel has five input arrays and two output arrays. By using this technique, all threads are able to read data from each input array and write data to each output array consecutively allowing all memory accesses to be coalesced. Coalescing all accesses without using shared memory reduces the number of instructions in the hand-optimized version. As a result, the performance of the hand-written program is 1.3x better than Sponge's generated code.

This input/output buffer re-mapping is not currently done in Sponge because StreamIt does not support actors with multi-inputs and multi-outputs streams. All input and output streams between StreamIt actors are through a single shared buffer between the actors. Future work will try to represent these multiple input/output streams in StreamIt so the compiler can detect such cases and improve memory layout for GPUs.

5.2 Histogram

The *histogram* benchmark computes the distribution of pixel intensities within an image. *Histogram* is implemented using a technique called stream reduction, which is common in many GPU applications. Each phase of stream reduction removes some elements of input data, performs computation on them, and sends the results as a new input to the next phase. The *histogram* benchmark has several phases. In the first phase, the input data array is divided into fixed size blocks. In the second phase, a sub-histogram for each block is computed. In the final phase, all the sub-histograms are collated into a single histogram.

A StreamIt graph of stream reduction is shown in Figure 12. The number of actors in these type of benchmarks is data-size dependent, therefore, as the size of the input data grows, the number of phases increases and the overhead of launching the kernels becomes dominant. Sponge can fuse all of these phases together but the final actor would have a large *pop* rate. Since the *pop* rate of this actor is very large, it is not possible to use the limited shared memory to coalesce its memory accesses. In both cases, the large number of kernels and the uncoalesced memory accesses result in degraded performance.

In the hand-optimized CUDA implementation, there is only one kernel for all of the phases of the reduction but the number of threads that do the actual work in each phase is different. As a result, the hand-written CUDA histogram benchmark outperforms Sponge's generate CUDA code by 5x.

We would like to enhance the performance of Sponge in this type of benchmarks by detecting the stream reduction subgraph in the compiler and replacing them with one specialized stream reduction kernel that mimics the behavior of the hand-optimized CUDA.

6. Related Work

The most common language GPU programmers use to write CUDA code is "C for CUDA" (C with NVIDIA extensions and certain restrictions). Tuning these C like programs is highly challenging because managing the amount of on-chip memory used per thread, the total number of threads per multiprocessor, and the pattern of off-chip memory accesses are some of the problems that developer need to solve manually to achieve good performance [23]. To alleviate this burden, recent studies has been done to automatically manage these parameters in CUDA programs.

One study, closely related to Sponge, is the optimizing compiler introduced by Udupa et al. [27]. They compile stream programs for GPUs using software pipelining techniques. In the software pipelining approach, different actors from different iterations

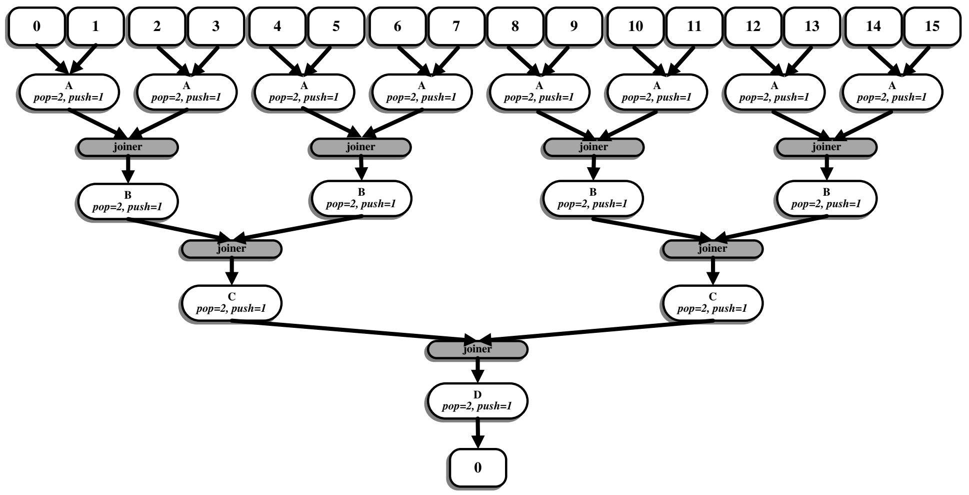


Figure 12: *This graph shows the stream graph of a generic stream reduction kernel.*

are simultaneously processed. Their technique, though promising, does not perform well on GPUs because it exploits task-level parallelism and is not able to exploit the massive amount of data-level parallelism power of GPUs. There has been recent work [29] on GPU compilation for memory optimization and parallelism management. The input to this compiler is a naive GPU kernel function and their compiler analyzes the code and generates optimized CUDA code. This work is distinctively different from this work because Sponge is able to exploit the information in the high level stream graph and perform kernel-level optimizations specific to StreamIt, such as graph restructuring, and then apply lower optimizations , such as memory and thread hierarchy management.

CUDA-Lite [30] is another compilation framework that takes naive GPU kernel functions as input and tries to coalesce all memory accesses by using shared memory. Programmers need to provide annotations describing certain properties of data structures and code regions designated for GPU execution. Our work is different because Sponge does not need any annotations. Sponge also uses shared memory to coalesce memory accesses and can maximize the utilization of various resources on GPUs, such as registers. Another difference is that when the size of shared memory limits the number of worker threads, Sponge is able to insert helper threads to accelerate the transferring of data between global and shared memory. hiCUDA [8] is a high level directive based compiler framework for CUDA programming where programmers need to insert directives to define the boundaries of the kernel function into sequential C code. Another work in the area of automatic CUDA generation is [16]. The Authors in this work generate optimized CUDA programs from OpenMP programs. They do not use shared memory in their compiler for coalescing memory accesses. Hong et al. [9] propose an analytical performance model for GPUs that compilers can use to predict the behavior of their generated code. Fung et al. [4] regroup threads into new warps to minimize the number of divergent warps. Chen et al. [2] use communication and computation threads to overlap the data exchange of the boundary nodes between adjacent thread blocks. This is fundamentally different from what Sponge achieves using helper threads by performing parallel prefetching of data.

MCUDA [24] tries to compile CUDA programs for a conventional shared memory architecture. MCUDA can be used to increase the performance of traditional shared memory parallel systems using CUDA optimization techniques. With the stream programming model, it is possible to use architecture specific optimizations for a wide range of architectures. Researchers have already proposed ways to map and optimize synchronous data-flow languages to SIMD engines [12], distributed shared memory systems [14], and also field programmable gate arrays [10].

Performing runtime re-compilation of GPU binaries for adapting code to different targets is another approach that can provide portability across GPUs. OpenCL [13] is one the approaches taken by industry to achieve portability. We believe OpenCL in its current form suffers from the same inefficiencies as CUDA and does not provide an architecture independent solution.

There is a large body of literature that deals with exploiting parallelism in streaming codes for better performance. The most recent and relevant works include compilation of new streaming languages such as StreamIt, Brook [1], Sequoia [3], and Cg [18] to multi-cores or data-parallel architectures. For example, Gordon et al. [6] and [5] perform stream graph refinements to statically determine the best mapping of a StreamIt program to a multi-core CPU. Liao et al. applies classic affine partitioning techniques to exploit the properties of stream operators [28]. There is also a rich history of scheduling and resource allocation techniques developed in Ptolemy that make fundamental contributions to stream-scheduling (e.g., [7, 22]). In a recent work [25], the authors talk about the usefulness of different features of StreamIt to a wide range of streaming applications. Several works, such as [11], propose techniques to dynamically recompile streaming application based on availability of resources in heterogeneous system. Sponge can be a complementary addition to these works as GPUs are becoming a commodity in heterogeneous systems.

7. Conclusion

Heterogeneous systems, where sequential work is done on traditional processors and parallelizable work is offloaded to a specialized computing engine, will be ubiquitous in the future. Among the different solutions that can take advantage of this parallelism, GPUs are the most popular solution and have been shown to provide significant performance, power efficiency and cost benefits for general purpose computing in highly-parallel computing domains. GPUs achieve their high performance and efficiency by providing a massively parallel architecture with hundreds of in-order cores and exposing parallelism mechanism and also the memory hierarchy to the programmer. One key to maximizing the performance in these future heterogeneous systems will be to efficiently utilize not only the host processor, but also the GPU.

While GPUs provide a very desirable target platform for accelerating parallel workloads, their programming complexity poses a significant challenge to application developers. Languages, such as CUDA, alleviate the complexity problem to some extent but fail at abstracting the underlying GPU architecture. Therefore, managing the amount of on-chip memory used per thread, the total number of threads per multiprocessor, and the pattern of off-chip memory accesses are problems that developers still need to manage in order to maximize GPU utilization.

In this work, we propose Sponge; a streaming compiler for the StreamIt language that is capable of performing an array of optimizations on stream graphs and generate efficient CUDA code for GPUs. Optimizations in Sponge facilitate a write-once software paradigm where programmers can rely on the compiler to automatically create customized CUDA for a wide variety of GPU targets. The optimizations in Sponge improve the performance compared to naive CUDA implementations by an average of 3.2x. Finally, as a case study, we compare the performance and implementation of two hand-optimized CUDA benchmarks, Black-Scholes and Histogram. For Black-Scholes, Sponge is able to achieve within 30% of the performance of the hand-optimized CUDA code. Future work on Sponge will improve automatic detection of certain memory layout characteristics and stream graph representations that are currently not supported.

Acknowledgement

Much gratitude goes to the anonymous referees who provided excellent feedback on this work. We also thank Todd Mowry for shepherding this paper. This research was supported by ARM Ltd. and the National Science Foundation under grants CNS-0964478 and CCF-0916689.

References

[1] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.

[2] J. Chen, Z. Huang, F. Su, J.-K. Peir, J. Ho, and L. Peng. Weak execution ordering - exploiting iterative methods on many-core gpus. In *Proc. of the 2010 IEEE Symposium on Performance Analysis of Systems and Software*, pages 154–163, 2010.

[3] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, 2006.

[4] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 407–420, 2007.

[5] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, 2006.

[6] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, Oct. 2002.

[7] S. Ha and E. A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, 40(11):1225–1238, 1991.

[8] T. Han and T. Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, (99):1–1, 2010.

[9] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 152–163, 2009.

[10] A. Hormati, M. Kudlur, D. Bacon, S. Mahlke, and R. Rabbah. Optimus: Efficient realization of streaming applications on FPGAs. In *Proc. of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 41–50, Oct. 2008.

[11] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proc. of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 214–223, 2009.

[12] A. H. Hormati, Y. Choi, M. Woh, M. Kudlur, T. Mudge, and S. Mahlke. Macross: Macro-simdization of streaming applications. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 285–296, 2010.

[13] KHRONOS Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2010.

[14] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of the '08 Conference on Programming Language Design and Implementation*, pages 114–124, June 2008.

[15] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[16] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 101–110, 2009.

[17] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 451–460, 2010.

[18] W. Mark, R. Glanville, K. Akeley, and J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *Proc. of the* 30^{th} *International Conference on Computer Graphics and Interactive Techniques*, pages 893–907, July 2003.

[19] NVIDIA. *CUDA Programming Guide*, June 2007. http://developer.download.nvidia.com/compute/cuda.

[20] NVIDIA. Fermi: Nvidias next generation cuda compute architecture, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[21] NVIDIA. Gpus are only up to 14 times faster than cpus says intel, 2010. http://blogs.nvidia.com/ntersect/2010/06/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel.html.

[22] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs. Technical Report UCB/ERL M95/36, University of California, Berkeley, May 1995.

[23] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008.

[24] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 16–30, 2008.

[25] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*, page To Appear, 2010.

[26] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002.

[27] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *Proc. of the 2009 International Symposium on Code Generation and Optimization*, pages 200–209, 2009.

[28] S. wei Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for brook streaming applications on multiprocessors. *Proc. of the 2006 International Symposium on Code Generation and Optimization*, 0(1):196–207, 2006.

[29] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 86–97, 2010.

[30] S. zee Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu. Cuda-lite: Reducing gpu programming complexity. In *Proc. of the 21st Workshop on Languages and Compilers for Parallel Computing*, pages 1–15, 2008.