Adaptive Input-aware Compilation for Graphics Engines

Mehrzad Samadi
University of Michigan, Ann Arbor
mehrzads@umich.edu

Amir Hormati
Microsoft Research, Redmond
amir.hormati@microsoft.com

Mojtaba Mehrara
NVIDIA Research, Santa Clara
mmehrara@nvidia.com

Janghaeng Lee
University of Michigan, Ann Arbor
jhaeng@umich.edu

Scott Mahlke
University of Michigan, Ann Arbor
mahlke@umich.edu

Abstract

While graphics processing units (GPUs) provide low-cost and efficient platforms for accelerating high performance computations, the tedious process of performance tuning required to optimize applications is an obstacle to wider adoption of GPUs. In addition to the programmability challenges posed by GPU's complex memory hierarchy and parallelism model, a well-known application design problem is target portability across different GPUs. However, even for a single GPU target, changing a program's input characteristics can make an already-optimized implementation of a program perform poorly. In this work, we propose Adaptic, an adaptive input-aware compilation system to tackle this important, yet overlooked, input portability problem. Using this system, programmers develop their applications in a high-level streaming language and let Adaptic undertake the difficult task of input portable optimizations and code generation. Several input-aware optimizations are introduced to make efficient use of the memory hierarchy and customize thread composition. At runtime, a properly optimized version of the application is executed based on the actual program input. We perform a head-to-head comparison between the Adaptic generated and hand-optimized CUDA programs. The results show that Adaptic is capable of generating codes that can perform on par with their hand-optimized counterparts over certain input ranges and outperform them when the input falls out of the hand-optimized programs' "comfort zone". Furthermore, we show that input-aware results are sustainable across different GPU targets making it possible to write and optimize applications once and run them anywhere.

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

General Terms Design, Languages, Performance

Keywords Streaming, Compiler, GPU, Optimization, Portability

1. Introduction

GPUs are specialized hardware accelerators capable of rendering graphics much faster than conventional general-purpose processors. They are widely used in personal computers, tablets, mobile phones, and game consoles. Modern GPUs are not only efficient at manipulating computer graphics, but also are more effective than CPUs for algorithms where processing of large data blocks is done in parallel. This is mainly due to their highly parallel architecture. Recent works have shown that in optimistic cases, speedups of 100-300x [20], and in pessimistic cases, speedups of 2.5x [15], are achievable using modern GPUs compared to the latest CPUs.

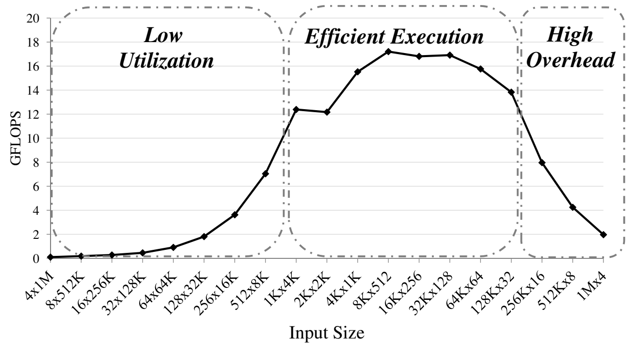


Figure 1: *Performance of the transposed matrix vector multiplication benchmark from the CUBLAS library on an NVIDIA Tesla C2050. The X-axis shows the input dimensions in number of rows x number of columns format.*

While GPUs provide inexpensive, highly parallel hardware for accelerating parallel workloads, the programming complexity remains a significant challenge for application developers. Developing programs to effectively utilize GPU's massive compute power and memory bandwidth requires a thorough understanding of the application and details of the underlying architecture. Graphics chip manufacturers, such as NVIDIA and AMD, have tried to alleviate part of the complexity by introducing new programming models, such as CUDA and OpenCL. Although these models abstract the underlying GPU architecture by providing unified processing interfaces, developers still need to deal with many problems such as managing the amount of on-chip memory used per thread, total number of threads per multiprocessor, and the off-chip memory access pattern in order to maximize GPU utilization and application performance [24]. Therefore, programmers must manually perform a tedious cycle of performance tuning to achieve the desired performance.

Many prior efforts have tried to address this programmability challenge mainly along three interrelated angles. The works in [2, 3, 12, 14] provide high-level abstractions at the language level to enable easier expression of algorithms. These abstractions are later used by the compiler to generate efficient binaries for GPUs. Adding annotations to current models (CUDA or OpenCL) or popular languages (C or Fortran) to guide compiler optimizations is another method used in [8, 31, 34]. Finally, works in [1, 25, 33] try to automatically generate optimized code from a basic, possibly poorly performing, parallel or sequential implementation of an application.

The hard problem of finding the optimal implementation of an algorithm on a single GPU target is further complicated when attempting to create software that can be run efficiently on multiple GPU architectures. For example, NVIDIA GPUs have different architectural parameters, such as number of registers and size of shared memory, that can make an implementation which is optimal for one

PLDI'12, June 11–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1205-9/12/06. . . $10.00

architecture sub-optimal for another. The situation is even worse if the goal is to have an optimal implementation for GPUs across multiple vendors. We call this issue the *target portability* problem.

The portability issue is not specific to moving applications across different GPU targets. Even for a fixed GPU target, changing the problem size and dimensions can make a specific implementation of an algorithm sub-optimal, resulting in poor performance portability. Figure 1 illustrates this issue for the transposed matrix vector multiplication (TMV) benchmark from the CUBLAS library . All input matrices have the same number of elements but arranged in different shapes. The benchmark performs consistently between 12 and 17 GFLOPs over the input dimension range of 1Kx4K to 128Kx32 on an NVIDIA Tesla C2050 GPU. However, when input dimensions fall out of this range, the performance degrades rapidly by up to a factor of more than 20x. The main reason for this effect is that the number of blocks and threads in the application are set based on the number of rows and columns in the input matrix. Therefore, this benchmark works efficiently for a certain range of these values and for other input dimensions, either there is not enough blocks to run in parallel and hide memory latency (towards the left end of X-axis in the figure), or the data chunk that each block is operating on is too small to amortize the overhead of parallel block execution (towards the right end of X-axis in the figure). We call this the *input portability* problem.

There are various reasons for this problem in GPUs such as unbalanced workload across processors, excessive number of threads, and inefficient usage of local or off-chip memory bandwidth. Unbalanced workloads occur when a kernel has a small number of blocks causing several processors to be idle during execution, which leads to under-utilization of GPU resources and poor performance. Excessive number of threads result in sequential thread execution due to lack of enough resources in the GPU to run all threads in parallel. Finally, inefficient memory bandwidth usage can be due to the pattern of memory accesses which are determined based on the size or dimensions of the input data in some programs. Therefore, memory optimizations must be adapted based on the input to efficiently utilize the memory bandwidth.

One solution to the input portability problem is to have the programmer design and develop different algorithms for each input range and size. However, this would impose a considerable implementation and verification overhead as applications become larger, more complex, and need to work across a wide range of inputs. For instance, as shown later, five distinct kernel implementations are needed to efficiently utilize the processing power of a GPU across the complete input spectrum in the TMV benchmark. Multi-kernel applications complicate matters even more as programmers must deal with the cross-product of choices for each kernel as the input is varied. Clearly, automatic tools will become essential to guarantee high performance across various input dimensions.

In this work, we focus on tackling the input portability problem while providing GPU target portability. We employ a high-level streaming programming model to express target algorithms. This model provides explicit communication between various program kernels and its structured and constrained memory access lets the compiler make intelligent optimization decisions without having to worry about dependencies between kernels. An *adaptive input-aware compilation system*, called *Adaptic*, is proposed that is capable of automatically generating optimized CUDA code for a wide range of input sizes and dimensions from a high-level algorithm description. Adaptic decomposes the problem space based on the input size into discrete scenarios and creates a customized implementation for each scenario. Decomposition and customization are accomplished through a suite of optimizations that include a set of memory optimizations to coalesce memory access patterns employed by the high-level streaming model and to efficiently execute algorithms that access several neighboring memory locations at the same time. In addition, a group of optimizations are introduced to effectively break up the work in large program segments for efficient execution across many threads and blocks. Finally, two optimizations are introduced to combine the work of two segments so that execution overhead can be reduced.

An enhanced version of the performance model introduced in [10] is employed to predict application behavior for each range of input size and dimensions. Based on these predictions, optimizations are applied selectively by the compiler. At runtime, based on the provided input to the program, the best version of the generated code is selected and executed to maximize performance. This method frees application developers from the tedious task of fine-tuning and possibly changing the algorithm for each input range.

The specific contributions offered by this work are as follows:

- We propose a system that treats input portability as a first class programmability challenge for GPUs and provide means to solve it.
- We propose input-aware optimizations to overcome memory related performance deficiencies and break up the work fairly between working units based on the input size and dimensions.
- We develop an adaptive compilation and runtime system that optimizes performance for various input ranges by conforming to the user input and identifying and adjusting required optimizations.

2. Background

Exposed communication and an abundance of parallelism are the key features making stream programming a flexible and architecture-independent solution for parallel programming. In this paper, we employ a stream programming model based on Synchronous Data Flow (SDF) model. In SDF, computation is performed by *actors*, which are independent and isolated computational units, communicating only through data-flow buffers such as FIFOs. SDF, and its many variants, expose input and output processing rates of actors. This provides many optimization opportunities that can lead to efficient scheduling decisions for assignment of actors to cores, and allocation of buffers in local memories.

One way of writing streaming programs is to include all the computation performed in an actor inside a *work* method. This 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, including StreamIt [27], also provide non-destructive reads, called *peek*, that do not alter the state of the input buffer. In this work, we use the StreamIt programming language to implement streaming programs. StreamIt is an architecture-independent streaming language based on SDF and allows the programmer to algorithmically describe the computational graph. In StreamIt, actors can be organized hierarchically into *pipelines* (i.e., sequential composition), *split-joins* (i.e., parallel composition), and *feedback loops* (i.e., cyclic composition).

To ensure correct functionality in StreamIt programs, it is important to create a steady state schedule which involves rate-matching of the stream graph. There is a buffer between each two consecutive actors and its size is determined based on the program's input size and pop and push rates of previous actors. Rate-matching assigns a repetition number to each actor. In a StreamIt schedule, an actor is enclosed by a *for-loop* that iterates as many times as this repetition number.

Finally, since StreamIt programs are incognizant of input size and dimensions, Adaptic's input code is the same for all inputs but the output implementation will be different for various input sizes. The techniques that we propose in this paper are evaluated on StreamIt but are applicable to other streaming languages as well.

3. Adaptic Overview

The Adaptic compiler takes a platform-independent StreamIt program, ranges of its possible input size and dimension values, and the target GPU as input, and generates optimized CUDA code based on those ranges and the target. A StreamIt program consists of several actors that can be described as fine-grained jobs executed by each thread. Each actor in the stream graph is converted to a CUDA kernel with a number of threads and blocks. By performing

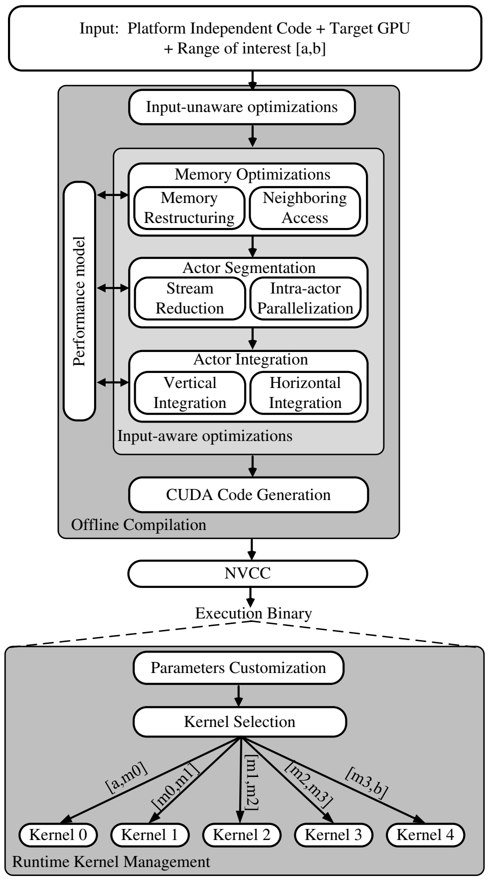


Figure 2: *Compilation Flow in Adaptic.*

input-aware stream compilation, Adaptic decides how many threads and blocks to assign to the CUDA kernel generated for each actor. Figure 2 shows the Adaptic's compilation flow which consists of four main components: baseline input-unaware optimizations, performance model, input-aware optimizations, and CUDA code generation. In addition, a matching runtime system selects appropriate kernels and sets their input parameters according to the program input at execution time. This section gives an overview of these four components as well as the runtime kernel management, while Section 4 details our proposed input-aware optimizations.

Input-unaware Optimizations: This step performs a set of input-unaware optimizations on the program and decides whether each actor should be executed on the CPU or GPU. This decision may be changed later by input-aware optimizations. Input-unaware optimizations are similar to those introduced in [12]. They include optimizations such as loop unrolling, data prefetching, and memory transfer acceleration. They can be used to generate CUDA code that is reasonably optimized and works for all input sizes, but its performance peaks for certain ranges of input and is suboptimal outside those ranges.

Performance Model: Adaptic relies on a high-level performance model to estimate the execution time of each kernel and to decide on using different optimizations for various problem sizes and GPU targets. This model is similar to the one described in [10], and classifies CUDA kernels into three categories of memory-bound, computation-bound, and latency-bound.

Memory-bound kernels have enough warps to efficiently hide the computation latency. Execution time of each warp is dominated by memory accesses. In computation-bound kernels, since most of the time is spent on computation, the execution time can be estimated as the total computation time. It should be noted that in these kernels, a large number of active warps is also assumed so that the scheduler would be able to hide memory access latencies with computation.

The last category, latency-bound kernels, are those that do not have enough active warps on each streaming multiprocessor (SM), and the scheduler cannot hide the latency of the computation or memory by switching between warps. Execution time of these latency-bound kernels is estimated by adding up the computation and memory access times. A kernel is latency-bound if there are not enough active warps for hiding the latency. There are two situations that lead to a small number of active warps: not enough data parallelism in the kernel or high shared resource consumption in each thread block.

In order to determine the type of each kernel, Adaptic counts the number of active warps on each SM. Based on this number and the target GPU, it determines whether the kernel is latency-bound or not. If not, Adaptic treats that kernel as both memory-bound and computation-bound and calculates the corresponding execution cycles based on the number of coalesced and non-coalesced memory accesses, computation instructions and the number of synchronization points, all of which are dependent on the input and can be computed at compile time as a function of input size and dimensions. The maximum of these two numbers determines the final kernel category.

Based on these categories, The performance model estimates the execution time of a kernel both before and after applying each optimization as a function of input dimensions. The performance break-even points determine the dimensions at which the corresponding optimization should be enabled or disabled.

Input-aware Optimizations: During each input-aware optimization phase, its potential performance impact for all input ranges is estimated using the model. These input ranges are provided by previous input-aware phases. If the optimization is beneficial, it is added to the optimization list for the whole range. However, if the optimization is only suitable for a subset of that range, Adaptic divides the range into smaller subranges, and populates optimization lists for each new subrange accordingly. In other words, Adaptic divides up operating input ranges to subranges if necessary, and applies different optimizations to each subrange. Therefore, separate kernels should be later generated for these subranges.

Code Generation: At the end of the compilation flow, the code generation stage generates optimized CUDA kernels for each input range based on optimization lists constructed by the optimization phase and the performance model. Since the performance model uses the target specifications to make optimization decisions, code generation is different for different targets. In addition, necessary code for runtime kernel management is also generated by the code generation unit based on the kernels and their operating input ranges. All these codes are later translated to a binary using the native CUDA compiler.

Runtime Kernel Management: A runtime kernel management unit is developed to dynamically select a properly optimized kernel at runtime based on the program input. This unit also determines the values of parameters that should be passed to each kernel at launch time including the number of blocks, number of threads per block, and the size of allocated shared memory. In order to remove kernel management overhead at runtime, this unit is completely executed on the CPU during the initial data transfer from CPU to GPU.

4. Input-aware Optimizations

As mentioned in Section 1, several factors such as inefficient use of memory bandwidth, unbalanced workload across processors, and excessive number of threads lead to ineffectiveness of input-unaware optimizations in achieving high performance across different inputs. The goal of input-aware optimizations in this work is to deal with these inefficiencies.

Two *memory optimizations* are introduced in Section 4.1 to solve inefficient use of local or off-chip memory bandwidth. In addition, two other sets of optimizations, namely *actor segmentation* and *actor integration* are detailed in Sections 4.2 and 4.3 respectively to tackle both unbalanced processor workload and excessive number of threads in the context of streaming.

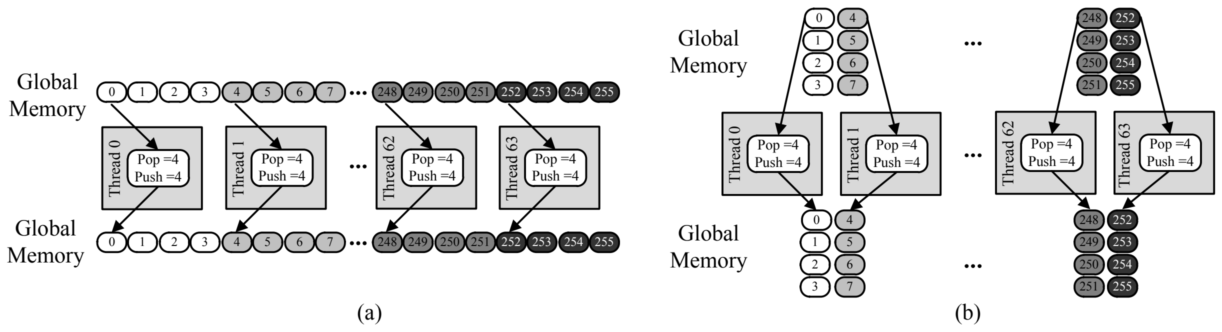


Figure 3: *Memory restructuring optimization. (a) Global memory access pattern of an actor with four pops and four pushes. Accesses are not coalesced because accessed addresses are not adjacent. (b) Access patterns after memory restructuring. Accessed addresses are adjacent at each point in time and accesses are all coalesced.*

4.1 Memory Optimizations

In this section, two memory optimizations, *memory restructuring* and *neighboring access optimization* are explained. These optimizations are useful for several classes of applications that are suitable for GPUs.

4.1.1 Memory Restructuring

One of the most effective ways to increase the performance of GPU applications is coalescing off-chip memory accesses. When all memory accesses of one warp are in a single cache line, the memory controller is able to coalesce all accesses into a single global memory access. Figure 3a illustrates how an actor with four pops and four pushes accesses global memory. In this example, each actor in each thread accesses four consecutive memory words. The first pop operations in threads 0 to 64 access memory word locations 0, 4, 8,. . . , 252, second pop operations access locations 1, 5, 9,. . . , 253, etc. Since these locations are not consecutive in memory, non-coalesced global memory accesses occur, leading to poor performance.

There are two ways to coalesce these memory accesses. One way is to transfer all data to the shared memory in a coalesced manner and since shared memory is accessible by all threads in a block, each thread can work on its own data. In this method, each thread fetches other threads' data from global memory as well as part of its own data. The same method can be applied for write-backs to global memory as well. All threads write their output to shared memory and then they transfer all data in a coalesced pattern to the global memory. Although using shared memory for coalescing accesses can improve performance, it has two shortcomings: number of threads is limited by the size of shared memory and the total number of instructions is increased due to address computations.

We use another method for coalescing accesses and that is to restructure the input array in a way that each pop access in all threads accesses consecutive elements of one row of the input in global memory. Figure 3b shows how this restructuring coalesces all memory accesses without using shared memory. This method has the advantage of minimizing the number of additional instructions and does not limit the number of threads by the size of shared memory. In addition, since this optimization is not using shared memory to coalesce off-chip memory accesses, shared memory can be utilized to store real shared data.

This optimization is not applicable when there are two or more actors with mismatching push and pop rates in the program. In those cases, rate matched buffers between kernels also have to be restructured, which involves extra write and reads from global memory, leading to poor performance.

However, as the work in [26] shows, since most consecutive actors in streaming benchmarks have matching rates, using memory restructuring would be beneficial. The CPU can restructure data at generation time and transfer it to the global memory of the GPU. The GPU launches kernels and when all of them are finished, the CPU reads back the output data. Due to the dependency of pop and push rates of some of the actors are on input size, this optimization can have different effects for various sizes.

In addition to coalescing global memory accesses, memory restructuring can also be applied to shared memory to remove bank conflicts. After applying this optimization, all threads access consecutive addresses in shared memory. Since adjacent addresses in shared memory belong to different shared memory banks, there would be no bank conflicts.

4.1.2 Neighboring Access Optimization

A common pattern in many applications is to access a point and its neighboring locations to calculate the data for that point. These benchmarks are very common in simulation benchmarks, for instance, the temperature of each point on a surface is computed based on the temperature of its neighbors. In streaming programs, non-destructive reads (peek) are used to read the neighbors' data while the address of the main point increases linearly in each iteration. Figure 4a shows an example StreamIt code for a *five-point stencil* actor that has neighboring access pattern and Figure 4b illustrates the access pattern for this code. In this example, each element is dependent on its top, bottom, right, and left elements. Each thread first reads all top elements, which are consecutive, leading to coalesced memory accesses. The same pattern holds for bottom, right, left and center elements. However, the main problem with this class of actors is excessive global memory accesses. For instance, accessing all top, bottom, left, right and center elements in each thread simply means accessing the whole input five times.

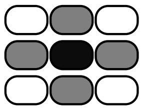
An efficient way to alleviate this problem is to use shared memory such that each block brings in one tile of data to shared memory and works on that. Since the data close to tile edges is needed for both neighboring tiles, tiles should be overlapping. These overlapping regions, called *halo*, are brought in for each tile at all four edges. Since branch divergence occurs only within a warp, both tile and halo widths should be multiples of warp size to make all accesses of each warp coalesced and prevent control flow divergence for address calculations.

Since halo regions are fetched for each tile, they should be as small as possible to minimize extra memory accesses. In order to decrease the overhead of extra memory accesses, Adaptic uses *super tiles*. In such a tile, the ratio of the halo size to the main tile size is decreased by merging several simple tiles. Each super tile is assigned to one block and each thread computes several output elements in different tiles. In this case, each block brings in a super tile from global memory to shared memory, performs the computation, and writes back the super tile to global memory.

Figure 5 shows a super tile assigned to a block in our example. Dark gray elements construct the main tiles while the light gray elements are halo parts. The number in each element indicates the thread index reading that element's address. In this example, warp size is set to 2 and there are 8 threads in each block. Each tile is 4x2 and by merging four tiles together, one super tile with 4x8 elements is formed. Since all width values should be multiples of warp size to maintain memory coalescing, the width of right and left halo parts in this example are set to 2. As a result, One of the halo values on each

```
5-Point Stencil(pop,peek:size, push:size)
  for (index=0; index<size; index++)
    if (not on edge)
      Top    = peek(index - width)
      Bottom = peek(index + width)
      Right  = peek(index + 1)
      Left   = peek(index - 1)
      Center = peek(index - 1)
    push(func(Top,Bottom,Right,Left,center))
```

(a)



(b)

Figure 4: *(a) An example StreamIt code of a five-point stencil actor. (b) Memory access pattern of this actor.*

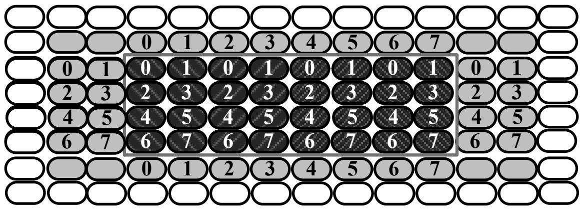


Figure 5: *A super tile assigned to one block. Dark gray addresses are main part and light gray parts are halo parts. Numbers in each small box indicates which thread reads this address.*

side (left and right) are not used in the computations for this super tile.

Increasing the size of super tiles leads to an increase in the allocated shared memory for each block, which in turn, could result in lower number of concurrent blocks executed on each GPU SM. Since this issue may change the type of kernel from computation-bound or memory-bound to latency-bound, the data size processed by each block should be chosen carefully.

For each application, Adaptic has to calculate the size and shape of a super tile. In general, the size of each super tile should not be more than the maximum shared memory per block, which is a constant value based on the target GPU. The super tile's size is dependent on the input size. For small input sizes it is beneficial to use smaller super tiles in order to have more blocks. Large super tiles are advantageous for large input sizes to reduce excessive memory accesses. Using the performance model and the input size, Adaptic can calculate the size for each super tile. Then, given the size, Adaptic needs to find the shape (width and height) of a super tile. For this purpose, Adaptic uses a reuse metric to maximize the number of served memory accesses while minimizing the size of extra halo regions. Adaptic uses the following reuse metric to find the optimal shape and size for each tile:

$$Reuse\_Metric = \frac{\sum_{Tile} Element\_Accesses}{Halo\_Size}$$

In this formula, $Element\_Accesses$ is the number of times each element in the shared memory is accessed during the computation of the whole output matrix, and the summation is taken over all elements in the tile. Since the best tile is the one with small halo region that can compute a large chunk of output, Adaptic chooses rectangular tiles with maximum possible $Reuse_Metric$s.

Once the size and shape of super tiles and halo regions are determined, the output CUDA code will be similar to the code shown in Figure 6. First, the kernel reads in the super tile and all its halos to the shared memory, after which synchronization makes shared memory visible to all threads. Subsequently, each block starts working on its own data residing in the shared memory to perform the computation and output the results.

4.2 Actor Segmentation

Optimizations in this category attempt to divide the job of one large actor between several threads/blocks to increase the performance. In order to have balanced workload across processors with efficient number of threads, this segmentation should be done based on the input size.

The two main optimizations in this category are *stream reduction* and *intra-actor parallelization*. Reduction is one of the important algorithms used in many GPU applications. The goal of *stream reduction* optimization is to efficiently translate reduction operations to CUDA in streaming programs. *Intra-actor parallelization*'s goal is to break the dependency between iterations of large loops and make them more amenable to execution on GPUs.

```
Neighboring-access Kernel <<<Blocks, threads>>>
 Top halo      → shared memory
 Bottom halo   → shared memory
 Right halo    → shared memory
 Left halo     → shared memory
 for tile in super tile
   tile → shared memory
 sync();
 Do computations and write results
```

Figure 6: *A generic neighboring access CUDA code. First, different halo parts and the super tile are moved from global to shared memory. Subsequently, computations are performed on the shared memory data.*

4.2.1 Stream Reduction

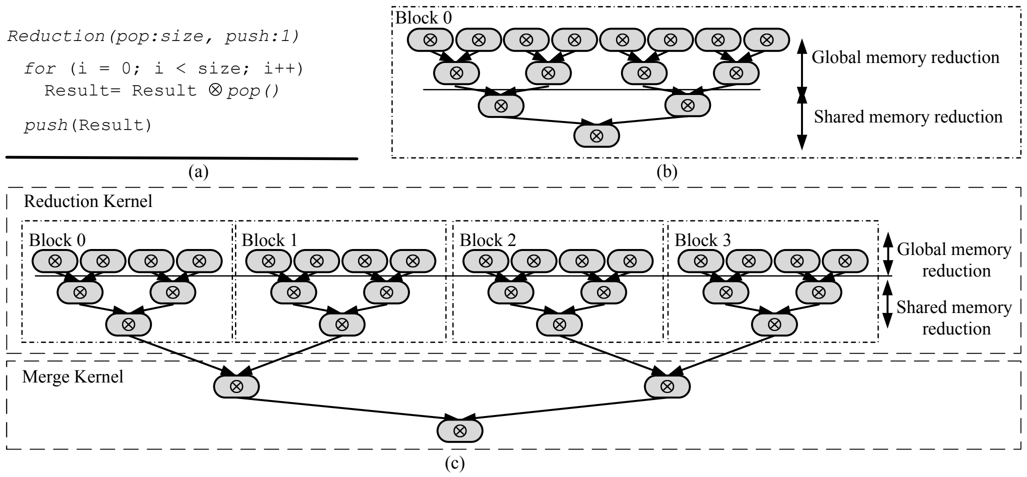
A reduction operation generally takes a large array as input, performs computations on it, and generates a single element as output. This operation is usually parallelized on GPUs using a tree-based approach, such that each level in the computation tree consumes the outputs from the previous level and produces the input for the next level. In uniform reduction, each tree level reduces the number of elements by a fixed factor and the last level outputs one element as the final result. The only condition for using this method is that the reduction operation needs to be associative and commutative.

A naive way of implementing the tree-based approach in a stream graph is to represent each tree node as an individual actor with small pop/push rates. Executing one kernel for each small actor would make the kernel launching overhead significant and degrade the performance dramatically. Another method of representing reduction is by using one filter that pops the whole input array and pushes the final result as shown in Figure 7a. The actor can not be translated to an efficient kernel due to the limited number of possible in-flight threads.

On the other hand, Adaptic automatically detects reduction operations in its streaming graph input using pattern matching. After this detection phase, it replaces the reduction actor with a highly optimized kernel in its output CUDA code based on the input size and the target GPU. This reduction kernel receives N_{arrays} different arrays with $N_{elements}$ elements each as input, and produces one element per array as output. Data is initially read from global memory, reduced and written to shared memory, and read again from shared memory and reduced to the final result for each array. In this work, we introduce two approaches for translating reduction actors to CUDA kernels.

When the array input size, $N_{elements}$, is small compared to the total number of input arrays (N_{arrays}) Adaptic produces a single reduction kernel in which each block computes the reduction output for one input array (Figure 7b). Thus, this kernel should be launched with N_{arrays} blocks. This approach is beneficial for large array counts so that Adaptic can launch enough blocks to fill up the resources during execution.

However, when the array input size ($N_{elements}$), is large compared to total number of input arrays (N_{arrays}), the reduction output for each array is computed individually by two kernels (Figure 7c). The first kernel, called the initial reduction kernel, chunks up the input array and lets each block reduce a different data chunk. The number of these blocks, $N_{initial\_blocks}$ is dependent on the value of $N_{elements}$ and the target GPU. Since there is no global synchronization between threads of different blocks, results of these blocks ($N_{initial\_blocks} * N_{arrays}$ elements) are written back to global memory. Subsequently, another kernel, called the merge kernel, is launched to merge the outputs from different blocks of the initial reduction kernel down to N_{arrays} elements. In the merge kernel, each block is used to compute the reduction output of one input array. Therefore, this kernel should be launched with N_{arrays} blocks.



```
Reduction(pop:size, push:1)

  for (i = 0; i < size; i++)
    Result= Result ⊗pop()

  push(Result)
```

Figure 7: *Stream reduction technique. (a) StreamIt code for a reduction actor. (b) Each block is responsible for computing output for one chunk of data in two phases. In the first phase, each thread reads from global memory and writes reduction output to the shared memory and in the second phase, shared memory data is reduced to one output element. (c) In the two kernel approach, different blocks of the first kernel work on different chunks of data and the second kernel reads all reduction kernel's output and compute final result.*

```
Initial Kernel Reduction<<<reductionBlocks, threads>>>
       /* Global memory reduction phase */
  Result = 0;
  numberOfThreads = BlockDim * gridDim;
  for ( index=tid; index<size; index+= numberOfThreads)
     Result = Result ⊗ Input[Index];

    SharedData[tid] = Result;

       /* Shared memory reduction phase */
  activeThreads = blockDim;
  while (activeThreads > WARP_SIZE){
    if (tid <activethreads)
L1    activeThreads /=2;
      sync();
      SharedData[tid] ⊗= SharedData[tid+activeThreads];
    }
  Stride = WARP_SIZE;
  if (tid < WARP_SIZE)
    while (stride > 1){
L2    sync();
      SharedData[tid] ⊗= SharedData[tid + stride];
      stride /=2;}
   if tid = 0
     Output[bid] = SharedData[0];
```

Figure 8: *The initial reduction kernel's CUDA code.*

Figure 8 shows Adaptic's resulting CUDA code for the initial reduction kernel. In the first phase, the input array in global memory is divided into chunks of data. Each thread computes the output for each chunk, and copies it to shared memory. The amount of shared memory usage in each block is equal to $Threads\_per\_Block * Element\_Size$. As discussed in Section 4.1.1, all global memory accesses are coalesced as a result of memory restructuring and there would be no bank conflicts in shared memory in this phase.

In the next phase, the results stored in shared memory are reduced in multiple steps to form the input to the merge kernel. At each step of this phase, the number of active threads performing reduction are reduced by half. Loop L1 in Figure 8 represents these steps. They continue until the number of active threads equals the number of threads in a single warp. At this point, reducing the number of threads any further would cause control-flow divergence and inferior performance. Therefore, we keep the number of active threads constant and just have some threads doing unnecessary computation (Loop L2 in Figure 8). It should be noted that after each step, synchronization is necessary to make shared memory changes visible to other threads. Finally, the thread with $tid = 0$ computes the final reduction result and writes it back to the global memory.

4.2.2 Intra-actor Parallelization

The goal of intra-actor parallelization is to find data parallelism in large actors. As mentioned before, it is difficult to generate optimized CUDA code for actors with large pop or push rates, consisting of loops with high trip counts. This optimization breaks these actors into individual iterations which are later efficiently mapped to the GPU. Using data flow analysis, Adaptic detects cross-iteration dependencies. If no dependency is found, Adaptic simply assigns each iteration to one thread and executes all iterations in parallel. It also replaces all induction variable uses with their correct value based on the thread index.

In some cases, Adaptic breaks the dependence between different iterations by eliminating recurrences. One common source of recurrence is accumulator variables. This happens when a loop contains an accumulator variable $count$ incremented by a constant C in every iteration ($count = count + C$). This accumulation causes cross-iteration dependencies in the loop, making thread assignment as described impossible. However, intra-actor parallelization technique breaks this dependence by changing the original accumulation construct to $count = initial\_value + induction\_variable * C$ and making all iterations independent.

In general, this optimization is able to remove all linear recurrence constructs and replace them by independent induction variable-based counterparts. This is similar to the induction variable substitution optimization that parallelizing compilers perform to break these recurrences and exploit loop level parallelism on CPUs.

4.3 Actor Integration

This optimization merges several actors or threads together to balance threads' workloads based on the input size in order to get the best performance. Two types of actor integration optimization are introduced in this paper. Vertical integration technique reduces off-chip memory traffic by storing intermediate results in the shared rather than global memory. Horizontal integration technique reduces off chip memory accesses and synchronization overhead and also lets the merged actors share instructions.

4.3.1 Vertical Integration

During this optimization, Adaptic vertically integrates some actors to improve performance by reducing memory accesses, removing kernel call overhead, and increasing instruction overlap. The reason for its effectiveness is that integrated actors can communicate through shared memory and there is no need to write back to the global off-chip memory. Also, integrating all actors together results in one kernel and global memory accesses of this one kernel are coalesced by the memory restructuring optimization. However, since input and output buffers of the middle actors in the integrated kernel are allocated in the shared memory, the number of active threads executing these actors are limited by the size of shared memory. This limitation often prevents Adaptic from integrating all actors together. Based on the performance model, Adaptic finds the best candidates for this optimizations. Since push and pop rates of some actors can be dependant on the input size, this optimization is beneficial for some ranges of input size.

Another optimization made possible after actor integration is replacing *transfer actors* with index translation. Transfer actors are the ones that do not perform any computation and only reorganize input buffer's data and write it to the output buffer. Since input and output buffers of the middle actors in integrated kernels are both allocated in the shared memory, there is no need to read the data from input buffer, shuffle it, and write it to the output buffer. This task can be done by index translation. Index translation gets thread indexes based on the transfer pattern, generates the new index pattern, and passes it to the next actor.

4.3.2 Horizontal Integration

The goal of horizontal integration is removing excessive computations or synchronizations by merging several threads or actors that can run in parallel. There are two kinds of horizontal integration techniques: horizontal actor integration and horizontal thread integration. In streaming languages, we use a duplicate splitter to allow different actors to work on the same data. In this case, instead of launching one kernel for each actor, one kernel is launched to do the job of all the actors working on the same data. Therefore, in addition to reducing kernel overheads, memory access and synchronization overheads are also reduced. For example, assume there is a program that needs maximum and summation of all elements in an array. Instead of running two kernels to compute these values, Adaptic launches one kernel to compute both. In this case, off-chip memory accesses and synchronizations only happen once instead of twice.

Horizontal thread integration merges several consecutive threads working on consecutive memory locations in one kernel. This method reduces the number of threads and blocks used by the kernel. Merged threads can share part of the computation that had to be done independently in each of the original threads and decrease the number of issued instructions. When the number of thread blocks is high, it is beneficial to use horizontal thread integration to reduce the number of threads and blocks and allow them to run in parallel. Otherwise it is better not to integrate threads and have more threads with less work to increase the possibility hiding memory latency by switching between threads.

5. Experiments

A set of benchmarks from the NVIDIA CUDA SDK and the CUBLAS library 3.2 are used to evaluate Adaptic. We developed StreamIt versions of these benchmarks, compiled them with Adaptic, and compared their performance with the original hand-optimized benchmarks. We also present three case studies to better demonstrate and explain the effectiveness of Adaptic's compilation algorithms. The first case study is performed on a CUBLAS benchmark to investigate the effect of our optimizations over a wide range of inputs. Then, we present two more case studies on real world applications, biconjugate gradient stabilized method and support vector machine [4], executed on two different GPUs, to demonstrate how Adaptic performs on larger programs with many actors and on different GPU targets. Adaptic compilation phases are implemented in the backend of the StreamIt compiler [27] and its C code generator is modified to generate CUDA code. Both Adaptic's output codes and the original benchmarks are compiled for execution on the GPU using NVIDIA nvcc 3.2. GCC 4.1 is used to generate the x86 binary for execution on the host processor. The target system has an Intel Xeon X5650 CPU and an NVIDIA Tesla C2050 GPU with 3GB GDDR5 global memory with NVIDIA driver 260.04. The other system used for experiments in Sections 5.2.2 and 5.2.3 has an Intel Core 2 Extreme CPU and an NVIDIA GeForce GTX 285 GPU with 2GB GDDR2 global memory.

5.1 Input Portability

In order to show how Adaptic handles portability across different input problem sizes, we set up seven different input sizes for each benchmark and compared their performance with the original CUDA code running with the same input sizes. It should be noted that these seven input sizes are chosen from the working range of the CUDA benchmarks because there are many sizes for which the SDK benchmarks would not operate correctly.

Figure 9 shows the results for eight CUDA benchmarks that were sensitive to changes in the input size, while results for input-insensitive benchmarks are discussed in Section 5.3. As can be seen, Adaptic-generated code is better than the hand-optimized CUDA code for all problem sizes in *Scalar Product*, *MonteCarlo*, *Ocean FFT*, and *Convolution Separable* from the SDK, and *Isamax/Isamin*, *Snrm2*, *Sasum*, and *Sdot* from CUBLAS. A combination of actor segmentation and actor integration were used to optimize all CUBLAS benchmarks. In addition to these optimizations, memory restructuring was applied to *Sdot*.

Sdot is computing the dot product of two vectors. For large vectors, using the two kernel reduction is beneficial, but for small sizes, in order to reduce kernel launch overhead, Adaptic uses the one kernel reduction. Using input-aware optimizations leads to upto 4.5x speedup in this benchmark compared to the original program. *Convolution Separable* has two actors, and processes data row-wise in one and column-wise in the other. Memory optimizations are effective for this benchmark as both of these two actors have neighboring memory access pattern. Therefore, as the input becomes smaller, Adaptic reduces the super tile sizes adaptively to retain the high number of blocks and, therefore, achieves better performance than the baseline hand-optimized code. *OceanFFT* also has a neighboring access actor and Adaptic uses different tile sizes to improve performance over the hand-optimized code. *Scalar Product* computes scalar products of pairs of vectors. The original benchmark uses the single kernel reduction, and it achieves good performance when there are many pairs of vectors in the input. However, for fewer pairs of vectors, it is better to use the whole GPU to compute the result for each pair. Using the two kernel reduction for those inputs, Adaptic is able to achieve upto 6x speedup compared to the original hand-optimized version.

MonteCarlo performs about the same as the original hand-optimized version. The reason is that the original benchmark already has two kernels performing the same task, but optimized for different ranges of input problem sizes. In other words, *MonteCarlo* has originally been developed in an input portable way. Therefore, the output of Adaptic is similar to the original version and the performance is the same for all sizes but it is generated automatically by the compiler.

Since Adaptic generates different kernels for some actors in the streaming program, the output binary size could be larger than the original binary optimized for one specific range. In our experiments including the case studies, Adaptic's output binaries were on average 1.4x and upto 2.5x larger than their input-unaware counterparts, which is quite reasonable considering the fact that some kernels could have upto five different versions for various input ranges. However, because each program also has kernels with one versions, the combination leads to this moderate code size increase.

These results further show the fact that our approach in Adaptic is able to adaptively generate optimized CUDA code for different problem sizes without any source code modifications.

5.2 Case studies

5.2.1 Transposed Matrix Vector multiplication

In this section, we look into the effects of our optimizations on the performance of the transposed matrix vector multiplication benchmark from CUBLAS over a wide range of input sizes and dimensions. As was mentioned in Section 1, the original benchmark cannot provide sustainable performance gains for different input dimensions. However, with the aid of input-aware optimizations, Adaptic is able to generate five different kernels with different structures, where each kernel is parametrized to get better performance for a

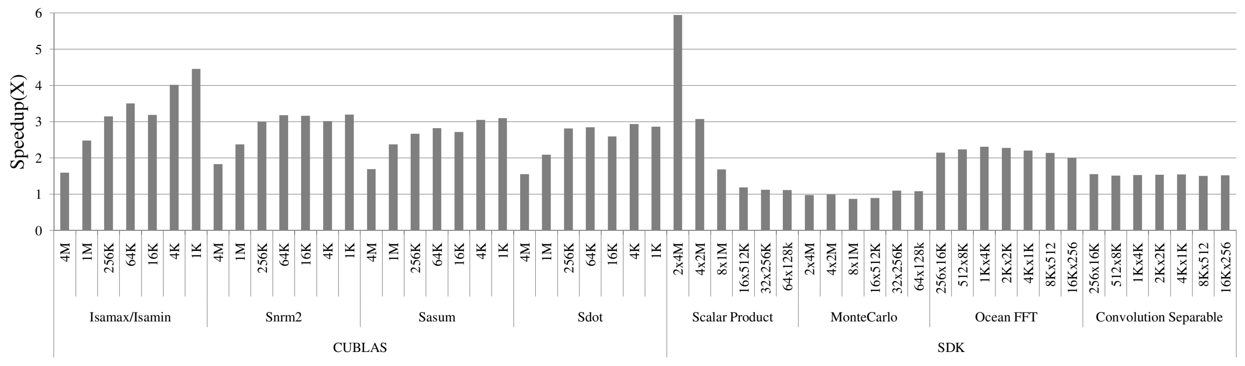


Figure 9: *Adaptic-generated code speedups normalized to the hand-optimized CUDA code for 7 different input sizes.*

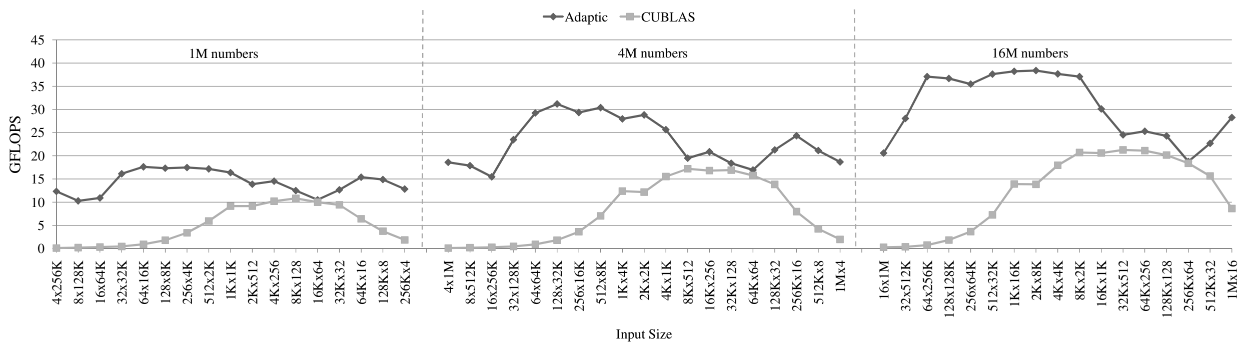


Figure 10: *Transposed matrix vector multiplication performance comparison of CUBLAS and Adaptic.*

specific range of input dimensions. At runtime the proper kernel is launched based on the program input.

In the first kernel, which is beneficial for matrices with many columns and few rows, Adaptic uses the two kernel version of reduction. For each row, one kernel is launched and the whole GPU is used to compute the dot product of one row with the input vector. The second kernel is a single-kernel reduction function where each block is responsible for one row. This kernel achieves its best performance for square matrices. In the third kernel, in addition to the single-kernel reduction function, by using horizontal thread integration, Adaptic adaptively merges several rows and each block is responsible for computing several dot products instead of one. This kernel is beneficial for matrices with more rows than columns. The fourth kernel is also similar to the single-kernel reduction, except that in its shared memory reduction phase, each thread is responsible for computing one output. The last kernel generated by Adaptic achieves its best performance for matrices with many rows and few columns. In this case, the size of each row is small and the corresponding actor has small pop rates. For this kind of actor, our baseline optimizations are effective in generating efficient code. Therefore, Adaptic does not need to add optimization to that. In this kernel, each thread is responsible for computing the dot product of a single row and the input vector.

Figure 10 compares the performance of this benchmark with Adaptic-generated code for three different matrix sizes over a range of matrix dimensions. As it can be seen, although for some input dimensions Adaptic's performance is really close to CUBLAS, for most of them Adaptic outperforms CUBLAS by a large margin.

5.2.2 Biconjugate gradient stabilized method

The biconjugate gradient stabilized method (BiCGSTAB) is an iterative method used for finding the numeral solution of nonsymmetric linear systems such as $Ax=B$ for x where A is a square matrix. This method has 11 linear steps that can be written easily with the CUBLAS library for GPUs. We wrote this program both in StreamIt and CUDA with CUBLAS functions and measured the performance of the two for different sizes of A. Figure 11 shows an in-depth com-

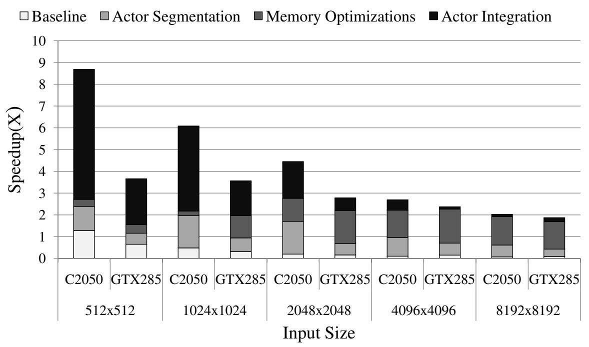


Figure 11: *Performance of the Adaptic-generated Biconjugate gradient stabilized method benchmark normalized to the CUBLAS implementation on two different GPU targets.*

parison and breakdown of the effects of Adaptic's individual optimizations on this benchmark for different input sizes across two GPU targets - NVIDIA Tesla C2050 and GTX285. The baseline in this figure is the generated code after only applying size-unaware optimizations. The Sgemv, Sdot, Sscal and Saxpy CUBLAS functions were used to implement the CUDA version of this benchmark. The problem of using the CUBLAS library is that the programmer should split each step into several sub-steps to be able to use CUBLAS functions. Execution of these sub-steps leads to more memory accesses and kernel launch overhead.

On the other hand, Adaptic merges all these sub-steps together and launches a single kernel for one step. As shown in Figure 11, most of the speedup for small sizes comes from the integration optimization. Since most of the execution time is spent in matrix vector multiplication for large sizes such as 8192x8192, the effect of integration is not as high for these sizes. However, actor segmentation that generates smaller actors and increases parallelism, and memory restructuring play more important roles in achieving better performance for larger sizes.

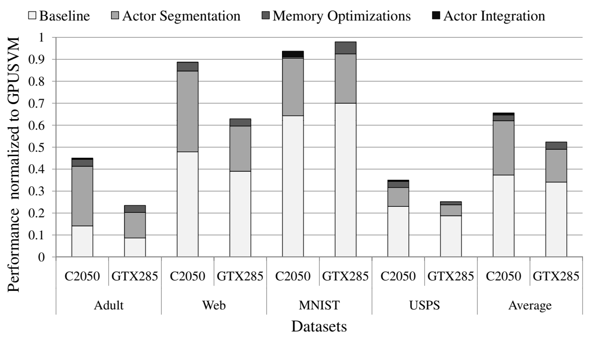


Figure 12: *Performance of the Adaptic-generated SVM training benchmark compared to the hand-optimized CUDA code in the GPUSVM implementation on two different GPU targets.*

5.2.3 Nonlinear Support Vector Machine Training

Support Vector Machines (SVMs) are used for analyzing and recognizing patterns in the input data. The standard two class SVM takes a set of input data and for each input predicts which class it belongs to among the two possible classes. This classification is based on a model, which is generated after training with a set of example inputs. Support vector machine training and classification are both very computationally intensive.

We implemented a StreamIt version of this algorithm based on the implementation in [4]. Figure 12 shows the performance of the Adaptic-generated code compared to the GPUSVM [4] hand-optimized CUDA code in this benchmark for four different input datasets. On average, Adaptic achieves 65% of the performance of the GPUSVM implementation. The reason for the large performance gap in Adult and USPS datasets is that GPUSVM performs an application-specific optimization where it utilizes unused regions of the GPU memory to cache the results of some heavy computations. In case those computations have to be performed again, it simply reads the results in from the memory. Therefore, for input sets which cause a lot of duplicate computations, including *Adult* and *USPS*, GPUSVM performs better than Adaptic-generated code.

In this program, unlike the previous example, actor integration is not very effective and most of the performance improvement comes from actor segmentation. On average, actor segmentation, memory restructuring, and actor integration improve the performance by 37%, 4%, and 1%, respectively.

5.3 Performance of Input Insensitive Applications

Although the main goal of Adaptic compiler is to maintain good performance across a wide range of inputs, it also performs well on the benchmarks that are not sensitive to input. Our experiments show that a combination of Adaptic optimizations makes the average performance of our compiler-generated code on par with the hand-optimized benchmarks, while writing StreamIt applications as the input to Adaptic involves much less effort by the programmer compared to the hand-optimized programs.

We ran Adaptic on a set of benchmarks from CUBLAS and the SDK (*BlackScholes*, *VectorAdd*, *Saxpy*, *Scopy*, *Sscal*, *Sswap*, and *Srot*, *DCT*, *QuasiRandomGenerator*, and *Histogram*), and on average the performance of Adaptic's output is within 5% of the original CUDA versions. This shows that Adaptic does not cause slowdowns for applications that are not sensitive to input size.

6. Related Work

The most common languages GPU programmers use to write GPU code are CUDA and OpenCL. Although these new languages partially alleviate the complexity of GPU programming, they do not provide an architecture-independent solution. There is an extensive literature investigating many alternative methods to support target portability.

Works in [2, 3, 7, 9, 12, 14, 21, 30] focus on generating optimized CUDA code from higher levels of abstraction. The Sponge compiler [12] compiles StreamIt programs and generates optimized CUDA to provide portability between different GPU targets. The work in [30] compiles stream programs for GPUs using software pipelining techniques. The work in [5] compiles Python programs for graphic processors. Copperhead [3] provides a nested set of parallel abstractions expressed in the Python programming language. Their compiler gets Python code as input and generates optimized CUDA code. It uses built-in functions of Python such as sort, scan, and reduce to abstract common CUDA program constructs. The work in [14] automatically generates optimized CUDA programs from OpenMP programs. MapCG [9] uses MapReduce framework to run high level programs on both multi-core CPUs and GPUs. Brook for GPUs [2] is one of the first papers about compilation for GPUs, which extends the C language to include simple data-parallel constructs. Compiling Matlab file to CUDA is also investigated in [21]. CnC_CUDA [7] use Intel's Concurrent Collections programming model to generate optimized CUDA code. All these works look into improving the programmability of GPUs, and in some cases, provide target portability. However, Adaptic provides portability across different inputs as well as GPU targets. In addition, Adaptic employs various input-aware optimizations and its output performance is comparable to hand written CUDA code.

Several other works have focused on automatically optimizing CUDA kernels [10, 33, 34]. The work in [33] performs GPU code compilation with a focus on memory optimizations 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 for various GPU targets. CUDA-Lite [34] is another compilation framework that takes naive GPU kernel functions as input and tries to coalesce all memory accesses by using shared memory. Hong et al. [10] propose an analytical performance model for GPUs that compilers can use to predict the behavior of their generated code. None of these works provide means to address the input portability problem.

The problem of input-aware optimizations has been studied in several previous works [17, 18, 28, 29]. The only work that tackles the input portability problem in the context of GPUs is introduced in [18]. In this work, programmers should provide optimizations' pragmas for the compiler and then compiler generates different programs with different optimizations based on those pragmas. Their approach is to run all these different versions of one program for different inputs and save all the results into a database. For each input, they check this database and find the best version and run it on the GPU.

There are other works that have focused on generating CUDA code from sequential input [1, 8, 16, 25, 31]. hiCUDA [8] is a high-level directive-based compiler framework for CUDA programming where programmers need to insert directives into sequential C code to define the boundaries of kernel functions. The work in [1] is an automatic code transformation system that generates CUDA code from input sequential C code without annotations for affine programs. In [31], by using C pragma preprocessor directives, programmers help compiler to generate efficient CUDA code. In [25], programmers use C# language and a library to write their programs and let the compiler generate efficient GPU code.The work in [16] proposes an extension to a Java JIT compiler that executes program on the GPU.

Gordon et al. [6] perform stream-graph refinements to statically determine the best mapping of a StreamIt program to a multi-core CPU. Researchers have also proposed ways to map and optimize synchronous data-flow languages to SIMD engines [11], distributed shared memory systems [13]. In a recent work [26], the authors talk about the usefulness of different features of StreamIt to a wide range of streaming applications.

Works in [22, 23, 32] map reduction to heterogeneous systems with GPUs. Mapping stencil loops to GPUs and tiling size tradeoff are also studied by [1] and [19]. However, Adaptic applies input-aware optimizations adaptively and more generally on streaming applications to provide input portability

7. Conclusion

GPUs provide an attractive platform for accelerating parallel workloads. However, their programming complexity poses a significant challenge to application developers. In addition, they have to deal with portability problems across both different targets and various inputs. While target portability has received a great deal of attention in the research community, the input portability problem has not been investigated before. This problem arises when a program optimized for a certain range of inputs, shows poor performance along different input ranges.

In this work, we proposed Adaptic, an adaptive input-aware compiler for GPUs. Using this compiler, programmers can implement their algorithms once using the high-level constructs of a streaming language and compile them to CUDA code for all possible input sizes and various GPUs targets. Adaptic, with the help of its input-aware optimizations, can generate highly-optimized GPU kernels to maintain high performance across different problem sizes. At runtime, Adaptic's runtime kernel management chooses the best performing kernel based on the input. Our results show that Adaptic's generated code has similar performance to the hand-optimized CUDA code over the original program's input comfort zone, while achieving upto 6x speedup when the input falls out of this range.

Acknowledgement

Much gratitude goes to the anonymous referees who provided excellent feedback on this work. This research was supported by ARM Ltd., the Gigascale Systems Research Center, and the National Science Foundation under grant CNS-0964478.

References

[1] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proc. of the 19th International Conference on Compiler Construction*, pages 244–263, 2010.

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

[3] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proc. of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 47–56, 2011.

[4] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proc. of the 25th International Conference on Machine learning*, pages 104–111, 2008.

[5] R. Garg and J. N. Amaral. Compiling python to a hybrid execution environment. In *Proc. of the 3rd Workshop on General Purpose Processing on Graphics Processing Units*, pages 19–30, 2010.

[6] 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.

[7] M. Grossman, A. Simion, Z. Budimli, and V. Sarkar. CnC-CUDA: Declarative Programming for GPUs. In *Proc. of the 23rd Workshop on Languages and Compilers for Parallel Computing*, pages 230–245, 2010.

[8] T. Han and T. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):52–61, 2010.

[9] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between CPU and GPU. In *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 217–226, 2010.

[10] 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.

[11] A. Hormati, Y. Choi, M. Woh, M. Kudlur, R. Rabbah, 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.

[12] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 381–392, 2011.

[13] 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.

[14] 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.

[15] 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.

[16] A. Leung, O. Lhoták, and G. Lashari. Automatic parallelization for graphics processing units. In *Proc. of the 7th International Conference on Principles and Practice of Programming in Java*, pages 91–100, 2009.

[17] X. Li, M. J. Garzarán, and D. Padua. A dynamically tuned sorting library. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 111–, 2004.

[18] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for GPU program optimizations. In *2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2009.

[19] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proc. of the 2009 International Conference on Supercomputing*, pages 256–265, 2009.

[20] 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.

[21] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *Proc. of the '11 Conference on Programming Language Design and Implementation*, pages 152–163, 2011.

[22] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proc. of the 2010 International Conference on Supercomputing*, pages 137–146, 2010.

[23] D. Roger, U. Assarsson, and N. Holzschuch. Efficient stream reduction on the GPU. In *Proc. of the 1st Workshop on General Purpose Processing on Graphics Processing Units*, pages 1–4, 2007.

[24] 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.

[25] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 325–335, 2006.

[26] 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*, pages 365–376, 2010.

[27] 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.

[28] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in stapl. In *Proc. of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–288, 2005.

[29] K. Tian, Y. Jiang, E. Z. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *Proceedings of the OOPSLA'10*, pages 125–139, 2010.

[30] 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.

[31] M. Wolfe. Implementing the PGI accelerator model. In *Proc. of the 3rd Workshop on General Purpose Processing on Graphics Processing Units*, pages 43–50, 2010.

[32] X.-L. Wu, N. Obeid, and W.-M. Hwu. Exploiting more parallelism from applications having generalized reductions on GPU architectures. In *Proc. of the 2010 10th International Conference on Computers and Information Technology*, pages 1175–1180, 2010.

[33] 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.

[34] 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.