

# A framework for efficient and scalable execution of domain-specific templates on GPUs

Narayanan Sundaram<sup>†‡</sup>, Anand Raghunathan<sup>†§</sup>, and Srimat T. Chakradhar<sup>†</sup>

<sup>†</sup> NEC Laboratories America, Princeton, NJ, USA

<sup>‡</sup> Department of EECS, University of California at Berkeley, CA, USA

<sup>§</sup> School of ECE, Purdue University, IN, USA

`narayans@eecs.berkeley.edu`, `raghunathan@purdue.edu`, `chak@nec-labs.com`

## Abstract

Graphics Processing Units (GPUs) have emerged as important players in the transition of the computing industry from sequential to multi- and many-core computing. We propose a software framework for execution of domain-specific parallel templates on GPUs, which simultaneously raises the abstraction level of GPU programming and ensures efficient execution with forward scalability to large data sizes and new GPU platforms. To achieve scalable and efficient GPU execution, our framework focuses on two critical problems that have been largely ignored in previous efforts - processing large data sets that do not fit within the GPU memory, and minimizing data transfers between the host and GPU. Our framework takes domain-specific parallel programming templates that are expressed as parallel operator graphs, and performs operator splitting, off-load unit identification, and scheduling of off-loaded computations and data transfers between the host and the GPU, to generate a highly optimized execution plan. Finally, a code generator produces a hybrid CPU/GPU program in accordance with the derived execution plan, that uses lower-level frameworks such as CUDA. We have applied the proposed framework to templates from the recognition domain, specifically edge detection kernels and convolutional neural networks that are commonly used in image and video analysis. We present results on two different GPU platforms from NVIDIA (a Tesla C870 GPU computing card and a GeForce 8800 graphics card) that demonstrate 1.7 - 7.8X performance improvements over already accelerated baseline GPU implementations. We also demonstrate scalability to input data sets and application memory footprints of 6GB and 17GB, respectively, on GPU platforms with only 768MB and 1.5GB of memory.

## 1. Introduction

Graphics processors, which have traditionally been used to execute only graphics workloads, have emerged as promising alternatives to accelerate a wide range of highly

parallel, compute-intensive applications. From an architectural perspective, GPUs have evolved from specialized application-specific circuits into relatively general-purpose architectures (called GPGPUs) that can be programmed to execute arbitrary computations. GPGPUs already integrate a large number of parallel processing cores (16-240) into a single chip, and could therefore be regarded as early instances of “many-core” processors. Finally, the economies of scale enabled by the mass-market usage of GPUs in desktop computers, laptops, and even mobile devices, makes them very attractive in terms of performance/price ratio. In the near future, most computing systems in use will already have GPGPUs that could be leveraged to perform other computations.

Due to the great promise of GPUs, recent years have witnessed myriad interesting applications being parallelized on them. Some examples include computational fluid dynamics, molecular simulations, biomedical image processing, securities modeling in finance, seismic data analysis for oil and gas exploration, image and video processing, and computer vision [2, 15]. However, many challenges remain to be addressed before the potential of GPUs can be truly harnessed on a broader scale. Despite significant advances in GPU programming, thanks to frameworks such as NVIDIA’s CUDA [15] and AMD’s Stream SDK [4], writing high-performance GPU programs remains a task that requires familiarity with the GPU architecture. The performance of a GPU program is impacted significantly and in complex ways by factors such as how the computation is organized into threads and groups of threads, register and on-chip shared memory usage, off-chip memory access characteristics, synchronization among threads on the GPU and with the host, and data transfer between the GPU memory and host memory. Due to these challenges, GPUs still remain inaccessible to domain experts who are used to programming in very high-level languages. The approach of application-specific GPU ports by GPU programming experts is not scalable in terms of programming effort.

We believe that the next significant step in the evolution towards GPU computing is the development of program-

ming frameworks that address the aforementioned challenges. We are not the first to recognize this need; recent efforts along this direction can be divided into three categories (i) libraries and run-times that implement data parallel programming models such as MapReduce on GPUs [1, 6, 11, 14], (ii) compilers and auto-tuners for GPUs [19, 18, 13, 16], and (iii) stream programming frameworks such as Peakstream, Rapidmind [17], and BrookGPU [5].

In this paper, we propose a framework for efficient and scalable execution of domain-specific parallel templates on GPUs. Our domain-specific templates include computations that can be represented as a graph of parallel operators. We believe that the use of domain-specific templates effectively bridges the abstraction gap between domain experts (algorithm designers) and current GPU programming frameworks (such as CUDA). Furthermore, we believe that the information embodied in these templates can be exploited to achieve GPU execution with high efficiency (performance) and scalability. An important aspect of our work is the focus on applications with data sizes that do not fit within GPU memory. Owing to the higher cost and speed of GPU memories, the amount of memory on most GPU platforms is both limited and fixed (*i.e.*, cannot be upgraded by the end-user). As a result, many interesting applications have data sets that do not fit into the GPU memory. Such applications pose a particular challenge to the programmer, who needs to explicitly break down the computations and associated data structures so as to fit within the limited GPU memory, specify the sequence of GPU operations and data transfers, and manage the allocation of GPU memory and the explicit copying of data back-and-forth between the host memory and the GPU memory to achieve correct and efficient execution.

The proposed framework consists of compilation steps that generate an optimized execution plan for a specified template, and a code generator that produces a hybrid CPU/GPU program in accordance with the generated execution plan. We consider domain-specific templates that are represented as graphs of parallel operators, where the memory footprints of the operators and their data-dependencies are statically defined, and their scaling behavior with respect to input data size is fully understood. We utilize techniques such as operator splitting, and offload and data transfer scheduling, to generate an efficient execution plan that is feasible for the given GPU memory capacity. Re-targeting to different data sizes and GPUs with different memory capacities is automatic and abstracted from the application programmer, who simply views the templates as parametrized APIs that implement specific algorithms.

We have implemented the proposed framework and applied it to algorithms from the recognition application domain. Specifically, we consider edge detection from images and convolutional neural networks (CNNs), which are

representative of pre-processing and core machine learning computations that are performed in recognition applications. Our framework allows applications written using GPU-independent domain-specific APIs to automatically execute on NVIDIA’s GPU platforms using the CUDA library. We evaluated the proposed framework on two platforms: (i) a Dell Precision T5400 workstation with two quad-core Intel Xeon E5405 processors and 8GB memory, and NVIDIA’s Tesla C870 GPU computing card (128-core GPU with 1.5GB memory), and (ii) an Intel Core 2 Duo processor with 8GB of memory and NVIDIA’s GeForce 8800 GTX graphics card (128-core GPU with 768MB memory). Our results demonstrate that the proposed methodology results in performance improvements from 1.7X to 7.8X on applications with memory footprints of 6GB and 17GB over a baseline manual GPU implementation. We also demonstrate automatic re-targeting of templates for different data sizes and target GPU platforms using the proposed framework.

## 2. Motivation

The overarching motivation for our work is to bridge the abstraction gap between domain-experts who are used to programming in very high-level languages and the state-of-the-art in GPU programming (such as CUDA), which is still relatively “low-level”. However, the adoption of any high-level programming framework depends on the performance of the programs that are executed using it. Achieving abstraction with poor performance is no more useful than achieving performance through low-level programming. With that in mind, we have focused on specific challenges related to performance when developing our framework. The first is how to execute computations on GPUs in the face of scaling data sizes, especially when the memory footprint does not fit in the physical GPU memory. The second related challenge is how to achieve high efficiency in offloading computations to the GPU by optimally utilizing its memory and minimizing data transfer costs between the host and GPU. Addressing these problems also improves the re-targetability of applications across different GPU platforms. Frameworks such as CUDA provide *functional portability* across different platforms from the same vendor, but performance can be significantly impacted by the different amounts of memory present in different GPU platforms (in some cases it might even be infeasible to execute the operations due to limited memory). Our framework enables application developers to also achieve *performance portability* to new GPU platforms and larger data sets. The issues that we address are complementary to other facets of code optimization for GPUs that have been considered in previous work [18, 19, 13].

In the rest of this section, we present examples to moti-

vate the need for scalability to data sizes that do not fit the GPU memory, and the associated need for optimizing data transfers in order to achieve efficient GPU execution.

## 2.1. Challenge 1: Scaling to data sizes larger than the GPU memory

In order to demonstrate the need for scalability to large data sizes, we consider an algorithm for edge detection from images. We apply this edge detection algorithm to extract edges from a high-resolution image that represents a histological micrograph of a tissue sample used for cancer diagnosis [7]. The original image and output edge map are shown in Figure 1(a). The algorithm is depicted as a data-flow graph of parallel operators in Figure 1(b), where the ellipses represent operators and rectangles represent the input, output, and intermediate data structures used in the algorithm.

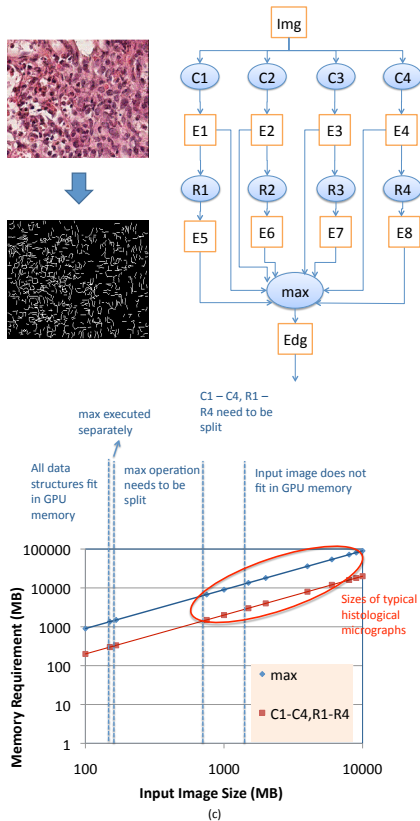


Figure 1: Edge detection in histological micrograph images (a) input/output images, (b) Algorithm used for edge detection, and (c) Memory requirements for the edge detection algorithm as a function of the input image size

Suppose that we would like to execute the edge detection algorithm shown in Figure 1(b) on the NVIDIA Tesla C870 GPU computing platform that has 1.5 GB of memory. Figure 1(c) presents the memory requirements for various

operators in the the edge detection algorithm as a function of the input image size. The *max* operator has the largest memory footprint (roughly nine times the input image size), while the other operators  $C1 - C4$ ,  $R1 - R4$  have a memory footprint of roughly twice the input image size. The graph is divided up into regions for which different strategies must be used in order to execute the edge detection algorithm within the limited memory of the C870 platform. These regions are separated by the vertical dashed lines, and the corresponding strategies used for executing the algorithm are indicated in the text above the graph.

- For image sizes of less than 150 MB, all the data structures associated with the edge detection algorithm fit in the GPU memory.
- For image sizes between 150 MB and 166.67 MB, the algorithm's memory footprint exceeds the GPU memory. However, the algorithm can be split into two parts - one executes the operators  $C1 - C4$  and  $R1 - R4$ , and the second executes the *max* operator.
- For image sizes between 166.67 MB and 750 MB, the *max* operator itself does not fit in the GPU memory and therefore it needs to be *split*.
- For image sizes between 750 MB and 1500 MB, the operators  $C1 - C4$  and  $R1 - R4$  also need to be split in addition to the *max* operator.
- For image sizes larger than 1500 MB, the input image itself does not fit in the GPU memory and therefore the entire algorithm needs to be divided to process the input image in chunks.

Clearly, the task of manually writing a *scalable* GPU implementation of the simple edge detection algorithm, *i.e.*, one that can handle input images of various sizes, is quite challenging. The application programmer needs to separately consider all the cases described above, determine the optimal execution strategy for each case, and combine the various scenarios into a single implementation using a framework such as CUDA. Debugging and maintaining such an implementation is likely to be quite a formidable task. In addition to that, problems arise when the code needs to be executed on another GPU platform that has a different memory capacity. Lower-level frameworks such as CUDA do not address the problem of automatically organizing computations so as to fit within limited GPU memory - this task is left to the application programmer.

Figure 1(c) also indicates the typical range of sizes of histological micrograph images that are encountered in the cancer diagnosis application. Clearly, the data sets are much larger than the memory capacities of even high-end GPU

computing platforms. Many other applications in Recognition and Mining workloads [8] process large data sets, making this challenge a common and important one to address.

Most of the GPU porting efforts to date typically assume that all the data fits within the GPU memory. While GPU memory capacities are certainly increasing, the rate of increase is much slower than the growth in many of the data sets that need to be processed. In order to achieve high internal memory bandwidth, GPUs use non-expandable GDDR memory that is packaged with the graphics processor in a single unit, and cannot be upgraded by the end user.

In summary, a critical need in GPU computing is to address the challenge of executing workloads whose memory footprint exceeds the available GPU memory. A related challenge is to ensure that programs written for today’s GPUs and data sets will work efficiently with the data sets and GPU platforms of the future with minimal programmer effort. Finally, it is also desirable that applications can be re-targeted to work on concurrently available GPU platforms that have different memory capacities (*e.g.*, high-end and low-end product variants). This problem has not been addressed in prior work on GPU computing.

An observation that falls out of the above example is that, as data sizes increase, a given computation needs to be divided up into smaller and smaller units in order to fit into the GPU memory, leading to increased data transfers between the CPU and GPU. This leads to the next challenge, namely minimizing the overheads of data transfer between the host and GPU.

## 2.2. Challenge 2: Minimizing data transfers for efficient GPU execution

One of the major performance limiting factors in GPU computing is the limited CPU to GPU communication bandwidth. State-of-the art GPU cards using the PCIe bus achieve a host-to-GPU bandwidth of around 1-2GB/s, which is much smaller than the internal memory bandwidth of GPU platforms which is over 64GB/s. This limitation is especially significant for applications that do not fit in the GPU memory and hence need to frequently transfer data structures between the host and GPU memory.

Figure 2 presents the breakdown of the time required to perform edge detection on an image of size  $8000 \times 8000$  with kernel matrices of varying sizes on the NVIDIA Tesla C870 platform. For various kernel matrix sizes ranging from  $2 \times 2$  to  $20 \times 20$ , the figure presents the breakdown of the total execution time into the time spent in data transfer to and from the GPU memory, and the time spent in computation on the GPU. The data transfer time varies from 30% of the overall execution time (for large kernel sizes, where more computation is performed per unit data) to 75% of the overall execution time for small kernel sizes. From our

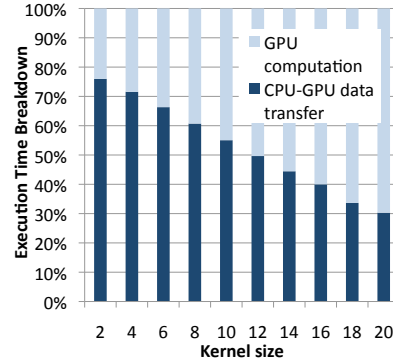


Figure 2: Execution time breakdown for executing image convolution operations with varying kernel matrix sizes on a GPU

experience with recognition applications such as edge detection and convolutional neural networks, we found that operations executed on the GPU generally spend up to 50% of the total runtime in data transfers between the CPU and GPU memory.

Current GPU programming models offer little help in managing data movement. The programmer needs to explicitly write code to copy data structures between the CPU and GPU, due to the fact that they have separate memory spaces. If all the data structures needed do not fit in the GPU memory, it is essential to manage the allocation of GPU memory over the duration of execution to store the most performance-critical data structures at any given time, to achieve the best possible execution efficiency.

We observe that the data transfer overheads in GPU execution are significantly influenced by the manner in which an application is broken down into computations that are atomically executed on the GPU, and the sequence of execution of these computations. In the context of domain-specific templates that are represented as graphs of parallel operators, operator scheduling has a very significant impact on the total volume of data transferred between the host and GPU.

To illustrate the large impact that scheduling has on data transfer overheads, we consider two alternative schedules for the edge detection algorithm that are shown in Figures 3(a) and 3(b). For the sake of illustration, we assume that the input image  $Im$  is of size 2 units. Note that the convolution, re-mapping, and  $max$  operators have been split into two in order to reduce their memory footprints. Therefore, all other data structures  $E1', E1'', \dots, E', E''$  are of size 1 unit each. We assume that the GPU memory capacity is 5 units. Consider the two different operator schedules shown in Figure 3(a) and 3(b). The schedule shown in Figure 3(a) executes the operators on the GPU in the sequence  $C_1 \rightarrow C_2 \rightarrow R'_1 \rightarrow R''_1 \rightarrow R'_2 \rightarrow R''_2 \rightarrow max_1 \rightarrow max_2$ . It can be shown that this schedule requires 15 units of data transfer between the CPU and the GPU. On the other hand,

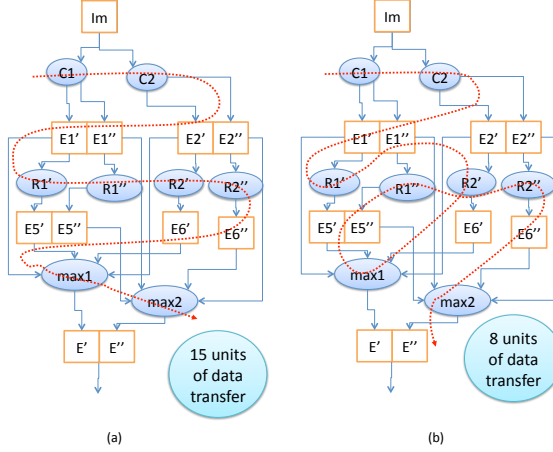


Figure 3: Two alternative schedules for edge detection that illustrate the impact of operator scheduling on data transfers

the schedule shown in Figure 3(b) that executes the operators in the order  $C_1 \rightarrow C_2 \rightarrow R'_1 \rightarrow R'_2 \rightarrow max_1 \rightarrow R''_1 \rightarrow R''_2 \rightarrow max_2$  requires only 8 units of data transfer. Since data transfers can take over 50% of the execution time, the schedule shown in Figure 3(b) will result in a GPU implementation that is much more efficient than the schedule shown in Figure 3(a). In general, the optimal schedule depends on the application characteristics (structure of the operator graph and sizes of data structures), and GPU memory capacity. For large applications (we consider operator graphs with thousands of operators), determining an efficient schedule of operators and data transfers is quite challenging for an application programmer.

In summary, GPU execution frameworks should address the challenges of scalability to data sizes that do not fit into the GPU memory and efficient offloading by minimizing the data transfer overheads. We discuss how our framework addresses these challenges in the next section.

### 3. GPU Execution Framework

In this section, we describe the proposed GPU execution framework that addresses the two aforementioned challenges for scalable and efficient execution. We present an overview of our framework in Subsection 3.1 and discuss how it overcomes both the challenges in the remainder of this section.

#### 3.1. Overview

Figure 4 presents the proposed framework for execution of domain-specific templates on GPUs.

The proposed framework takes as inputs a description of the domain-specific template and parameters that characterize the target GPU platform. It also assumes that an

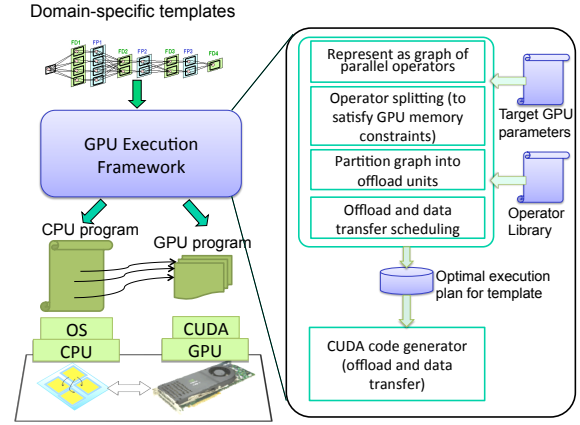


Figure 4: Proposed GPU Execution Framework

operator library that implements all the parallel operators is available. The framework generates an optimized execution plan for the template that specifies the exact sequence of offload operations and data transfers that are required in order to execute the template. The significant steps involved in the generation of the execution plan are as follows. The template is first represented as a parallel operator graph, wherein each vertex represents a parallel computation, and the directed edges between vertices represent the data dependencies. The next step identifies operators whose memory footprint exceed the GPU memory capacity, and performs operator splitting in order to split such operators<sup>1</sup>. The next step is to partition the operator graph into offload units, or sub-graphs that are atomically offloaded onto the GPU. Having coarser-grained offload units reduces synchronization overheads between the host and the GPU, however, the memory footprint may also increase and care must be taken to ensure that each offload unit can be individually executed within the available GPU memory. In our implementation, the individual operators are taken to be the offload units. The next step is to perform offload unit and data transfer scheduling. Here, the offload operations to the GPU are sequenced and a minimal set of data transfers are inferred. The scheduling is performed so as to minimize the total data transfer cost between the host and the GPU. Performing these two tasks (operator splitting and scheduling) simultaneously is difficult, and hence our framework handles them in two steps at the cost of optimality. Further details about the operator splitting and offload and data transfer scheduling steps are provided in subsections 3.2 and 3.3, respectively.

A code generator takes the execution plan generated by the previous steps, and generates a hybrid CPU/GPU pro-

<sup>1</sup> Arbitrary parallel operators are supported by the proposed framework, as long as their memory footprints are statically defined, and splitting rules are defined for operators that exceed the GPU's memory capacity.

gram that uses the desired GPU programming framework such as CUDA. The generated code and operator library can be compiled together with a larger application, and executed on top of a lower-level GPU run-time library such as CUDA.

### 3.2. Operator splitting: Achieving scalability with data size

The first challenge is to ensure that the template can be executed on the GPU regardless of its memory limitation. We assume that the operators are split-able *i.e.*, if the data needed for an operation does not fit in the GPU memory, it can be split (executed on several small portions of its input). Splitting enables us to execute arbitrary sized computations on the GPU, providing scalability. Data parallel operators provide an easy target for splitting. However, our framework can also handle other split-able, but not data parallel operators (*e.g.* convolution and reduction).

The operator splitting algorithm can be summarized as follows:

1. Compute the memory requirements of all operators (sum of sizes of data structures associated with each operator). Note that any operator whose memory requirements are larger than the available GPU memory cannot be executed without any modifications.
2. Split the operators whose memory requirements are greater than the GPU memory. This step ensures feasibility for all operators. When an operator is split, other operators that produce/receive data from the split operator also need to be modified.
3. Perform steps 1 & 2 until it is feasible to execute all operators on the GPU.

We consider convolutions, which are not strictly data parallel operations since they depend on a local neighborhood of points. This results in a need for more intelligent splitting that is dependent on the size of the convolution kernel. For example, a  $100 \times 100$  matrix convolved with a  $5 \times 5$  kernel matrix results in an output of size  $96 \times 96$  (ignoring borders). Splitting this operation into two must produce two  $100 \times 52$  input matrices (not  $100 \times 50$ ) and two  $96 \times 48$  output matrices. This size and offset computation can be done by traversing the split graph from the leaves to the root and inferring the sizes and offsets from the kernel and output sizes.

The ability to split operators can be taken for granted in the case of operators that are data parallel. For other operators, one could provide splitting rules, or hints to the framework to split the input/output data of an operator in specific ways. For example, a large matrix-matrix multiply

that does not fit in the GPU memory can be split by breaking up one of the input matrices and the output matrix. In the case of image convolutions, only the image matrix must be split. The convolution kernel matrix (which is also an input) should not be split. Even if an operator is not splittable, our framework is usable as long as this operator fits in the GPU memory.

### 3.3. Operator and data transfer scheduling

Once the operators are split such that every individual operation can be run on the GPU, they will have to be scheduled at a macro level to get the most efficient code. Finding the optimal schedule (in terms of minimal data transfers) given a template whose operators are individually schedulable is the key problem to be solved. We propose a heuristic method to solve the problem that uses a depth-first heuristic for scheduling the operators and a “latest time of use” based data transfer scheduling heuristic. We also model the problem formally as a Pseudo-Boolean optimization problem in Section 3.3.2. We discuss these methods in detail below.

#### 3.3.1 Heuristic solution

The data transfer optimization problem can be thought of as composed of two sub-problems - find a good operator schedule, and then, find the optimal data transfer schedule given this operator schedule.

Finding a good operator schedule is a difficult problem. This problem is similar to a data-flow scheduling problem. Since the aim is to maximize data reuse so that we need not transfer things back and forth between the CPU and GPU, we decided to adopt a *depth-first schedule* for the operators. In a depth-first schedule we try to schedule the entire sub-tree belonging to a child of a node before exploring its sibling. If a node cannot be scheduled due to precedence constraints (all its inputs are not ready), we backtrack to its parent and explore its other children. The drawback of the approach is that the operator schedule does not take into account the GPU memory limitations at all. While this heuristic can lead to reasonable results, there is scope for improvement by using information about the available GPU memory.

After a schedule for the operators is obtained, the data transfers can be scheduled. The only constraint to this problem is the amount of GPU memory available. This problem is similar to a cache replacement policy as we have a limited amount of fast memory where we would like to keep as much data as possible. We know that the optimal solution for the cache replacement problem is to replace cache lines that are going to be accessed furthest in the future. Based on this insight, we formulate a data transfer scheduling algorithm as follows:

1. Calculate the “latest time of use” for each data structure (since the operator schedule is known, this can be computed statically).
2. When a data structure needs to be brought into the GPU memory (*i.e.*, it is the input or output of the operator being executed at the current time step), and there is insufficient space, move the data structures that have the furthest “latest time of use” to the CPU until the new data structure can be accommodated.
3. Remove data eagerly from GPU memory *i.e.*, delete them immediately after they become unnecessary.

The solution generated by above algorithm will be optimal for a given operator schedule provided all the data structures are of the same size and are consumed exactly once. If all the data structures are of different sizes, the problem is equivalent to a bin-packing/knapsack problem which is proven to be NP-Complete. However, we have found that our heuristic works reasonably well in practice.

### 3.3.2 Formulation as a Pseudo-Boolean Optimization Problem

The problem of optimizing CPU-GPU data transfers can be written as a constraint satisfiability problem. In particular, it is possible to formulate it as a Pseudo-Boolean (PB) optimization problem. The Pseudo-Boolean optimization problem is a generalization of the satisfiability (SAT) problem. In a PB optimization problem, the variables are all Boolean, the constraints can be specified as linear equalities/inequalities and the objective function to be optimized is a linear function of the variables.

The variables used in our formulation are as follows:

- $x_{i,t}$  is 1 if operator  $i$  is executed at time step  $t$
- $g_{j,t}$  is 1 if data structure  $j$  is present in the GPU at time step  $t$
- $c_{j,t}$  is 1 if data structure  $j$  is present in the CPU at time step  $t$
- $Copy\_to\_GPU_{j,t}$  is 1 if data structure  $j$  has to be copied from the CPU to the GPU at time step  $t$
- $Copy\_to\_CPU_{j,t}$  is 1 if data structure  $j$  has to be copied from the GPU to the CPU at time step  $t$
- $done_{i,t}$  is 1 if operator  $i$  has been executed by time step  $t$
- $dead_{j,t}$  is 1 if data structure  $j$  is not needed after time step  $t$

The following constants are specific to the GPU platform and the template:

- $D_j$  is the size of the data structure  $j$
- $Total\_GPU\_Memory$  is the total amount of GPU memory present in the system
- $Output$  is the set of all the data structures that are outputs of the template (needed on the CPU)

- $IA_{i,j}$  is 1 if data structure  $j$  is an input to operator  $i$
- $OA_{i,j}$  is 1 if data structure  $j$  is an output of operator  $i$

Our PB formulation is given in Figure 5. Constraints (1-3) encode the precedence and scheduling requirements. There must be only one operation executing on the GPU at any given time. A task which is dependent on other tasks (data dependencies) can only execute after its dependencies are met. Constraint (4) specifies that at any time, the amount of data stored on the GPU cannot exceed the GPU memory. Constraints (5-8) encode the data movement and persistence on the GPU. Specifically, the input and output data required for an operation must exist on the GPU memory before it can be executed. If they do not exist, then they have to be copied to the GPU. For output data, enough space must be reserved before the execution of the operation. Constraints (9-10) specify similar properties for data in the CPU memory. Data resident on the CPU has to be invalidated if it is overwritten by a GPU operation. Data copied to the CPU or GPU remain there until moved, deleted or invalidated. Constraints (11-12) specify the initial conditions (all data resides on CPU, none on GPU). Constraint (13) gives the final condition (outputs must be in CPU memory). Constraints (14-19) specify data liveness. Data that is required in the future needs to be kept live (either in CPU or GPU memory). Otherwise, it can be deleted from the system.

We can solve this formulation using PB solvers like MinisAT+ [9]. However, the number of constraints in this formulation scale as  $O(N^2M)$  where  $N$  is the number of operators and  $M$  is the number of data structures. This method of solving it is feasible only for relatively small problems (up to few tens of operators). For problems containing hundreds or thousands of operators and data structures, solving the pseudo-Boolean optimization is practically infeasible. The heuristics mentioned in Section 3.3.1 are scalable, though may be suboptimal. When the operator schedule is known, the number of constraints in the data transfer scheduling problem scale as  $O(NM)$ . This problem is sufficiently large that it cannot be solved exactly using pseudo-Boolean solvers for large graphs.

Current GPUs have the ability to perform asynchronous data transfer and computation at the same time (as long as they are independent). This can be included in the formulation by changing the objective function to count only those transfers that involve data needed for the current computation. We did not overlap computation and communication in our experiments since the GPUs that we used did not support this capability.

The Pseudo-Boolean formulation ignores the fact that that GPU memory can get fragmented. In practice, the  $Total\_GPU\_Memory$  parameter in the formulation is set to a value less than the actual amount of GPU memory present in the system to account for fragmentation.

$minimize \sum_{j=1}^J \sum_{t=1}^N (Copy\_to\_CPU_{j,t} + Copy\_to\_GPU_{j,t}) \cdot D_j$		
(1)	$\forall t \sum_{i=1}^N x_{i,t} = 1$	Precedence & Scheduling
(2)	$\forall i \sum_{t=1}^N x_{i,t} = 1$	
(3)	$\forall i_1 \rightarrow i_2, t_1 > t_2, x_{i_1, t_1} + x_{i_2, t_2} \leq 1$	
(4)	$\forall t \sum_{j=1}^J g_{j,t} D_j \leq Total\_GPU\_Memory$	Memory
(5)	$\forall i, j, t [IA_{i,j} \vee OA_{i,j}] \wedge x_{i,t} \Rightarrow g_{j,t}$	GPU copy & persistence
(6)	$\forall i, j, t IA_{i,j} \wedge x_{i,t} \wedge \neg g_{j,t-1} \Rightarrow Copy\_to\_GPU_{j,t}$	
(7)	$\forall j, t Copy\_to\_GPU_{j,t} \Rightarrow g_{j,t}$	
(8)	$\forall j, t g_{j,t} \Rightarrow g_{j,t-1} \vee Copy\_to\_GPU_{j,t} \vee [\bigvee_{i=1}^N (OA_{i,j} \wedge x_{i,t})]$	
(9)	$\forall i, j, t OA_{i,j} \wedge x_{i,t} \wedge \neg Copy\_to\_CPU_{j,t+1} \Rightarrow \neg c_{j,t+1}$	CPU copy & persistence
(10)	$\forall j, t \neg c_{j,t} \wedge \neg Copy\_to\_CPU_{j,t+1} \Rightarrow \neg c_{j,t+1}$	
(11)	$\forall j c_{j,0} = 1$	Initial & Final conditions
(12)	$\forall j g_{j,0} = 0$	
(13)	$\forall j \in Output c_{j,N+1} = 1$	
(14)	$\forall i, t done_{i,t} \Leftrightarrow x_{i,t} \vee done_{i,t-1}$	Data liveness
(15)	$\forall i done_{i,0} = 0$	
(16)	$\forall j \in Output, t dead_{j,t} = 0$	
(17)	$\forall j \notin Output, t dead_{j,t+1} \Leftrightarrow dead_{j,t} \vee [\bigwedge_{i=1}^N (\neg IA_{i,j} \vee done_{i,t})]$	
(18)	$\forall j dead_{j,1} = 0$	
(19)	$\forall j, t \neg dead_{j,t} \Rightarrow c_{j,t} \vee g_{j,t}$	

Figure 5: Pseudo-Boolean Formulation of Offload and Data Transfer Scheduling

**Example:** To illustrate the scheduling of operators and data transfers, we consider the edge detection template shown in Figure 6(a). For simplicity of illustration, we once again assume that the input image is of size 2 units, all other data structures are of size 1 unit each, and the GPU memory capacity is 5 units. The optimal schedule obtained by solving the Pseudo-Boolean formulation presented in Section 3.3.2 is indicated in Figure 6(a). Figure 6(b) shows a detailed timeline with the actual sequence of data transfers and GPU offload operations. At any time, the data structures that are alive in the host and GPU memory are also indicated (the width of the boxes representing data structures are proportional to their size). The execution proceeds as follows:

- Initially, the input image is copied from the host memory to the GPU memory.
- Operator  $C_1$  is executed on the GPU, generating data structures  $E'_1$  and  $E''_1$  in the GPU memory.  $E'_1$  is copied back into the host memory in order to create enough space for the next operator.
- Operator  $C_2$  is executed on the GPU, generating  $E'_2$  and  $E''_2$  in the GPU memory. The input image  $Img$  is now deleted from GPU memory since it is no longer required, creating 2 units of space in the GPU memory<sup>2</sup>.

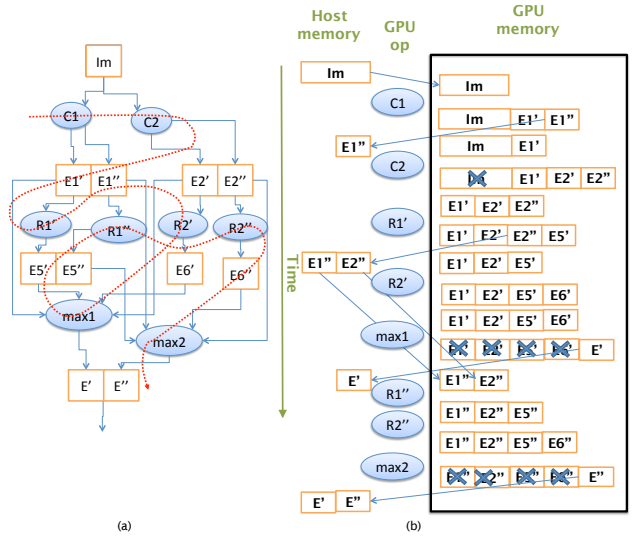


Figure 6: Operator and data transfer schedule (edge detection)

<sup>2</sup>The GPU memory occupied by  $Img$  may simply be released and reused by other data structures

- Similarly, all the other operators ( $R'_1, R'_2, max1, R''_1, R''_2$  and  $max2$ ) are also executed according to the schedule in Figure 6(b).

The execution sequence of operators on the GPU, and data transfers to and from the GPU memory, is referred to as an execution plan. In our framework, execution plans are used by a code generation framework to statically generate GPU code for the template that can be compiled and executed on the GPU using a lower-level framework such as CUDA. Alternatively, it is also possible to use a simple runtime library to orchestrate execution of the corresponding templates on the GPU.

## 4. Results

We present the results of using our framework on two different applications that use the templates we described earlier. We performed our experiments on two different systems. One system had a dual socket Quad core (Intel Xeon 2.00 GHz) as our primary CPU with 8 GB of main memory and an NVIDIA Tesla C870 as the GPU. The second system had an Intel Core 2 Duo 2.66 GHz with 8 GB of main memory and an NVIDIA GeForce 8800 GTX as the GPU. The Tesla C870 has 1.5 GB of memory while the GeForce 8800 GTX has 768 MB of memory. Both the GPUs have the same clock frequency (1.35 GHz) and degree of parallelism (128 cores) and differ only in the amount of memory. The two systems were running Redhat Linux and Ubuntu Linux, respectively. CUDA 2.0 was used for GPU programming.

For comparison purposes, we propose the following execution pattern as the baseline for GPU execution. For each operator, transfer input data to the GPU, perform the operation and copy the results back to the CPU immediately. There is no persistent storage in GPU memory. It allows any operator to execute on the GPU without any interference from other operators. However, this is suboptimal when all the temporary data structures fit the GPU memory (the optimal solution would be to move only the overall inputs and outputs). Currently, most GPU porting efforts implicitly make the assumption that the GPU memory is large enough to hold all the data. We are interested in large problems where the data does not fit the GPU memory.

### 4.1. Description of Templates

#### 4.1.1 Edge detection

Edge detection is one of the most important image processing operations that is performed as a pre-processing/feature extraction step in many different applications. Computationally, it involves convolving the input image with rotated

versions of an edge filter at different orientations and then combining the results by doing a reduction such as addition/max/max absolute value, *etc.* The general template for edge detection is given as follows :

$$edge\_map = find\_edges(Image, Kernel, num\_orientations, Combine\_op)$$

This is similar to the template shown in Figure 1 (where some convolutions are replaced by “remap” (R) operators). We obtained this template from a cancer detection application [7]. Edge detection alone contributes to about 30% of the total runtime of the application.

We performed edge detection using a  $16 \times 16$  sized edge filter at four different orientations (2 convolutions and 2 remaps) and combined the results using a Max operation as in Figure 1. The edge detection template is reasonably small so we can apply the pseudo-Boolean solver to find the optimal solution as described in Section 3.3.2.

We perform our experiments on both small ( $1000 \times 1000$ ) and large ( $10000 \times 10000$ ) input images to the template. The results are also shown in Tables 1 and 2.

#### 4.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are used extensively in many machine learning tasks like handwritten digit recognition, OCR, face detection, *etc.* We obtained our CNN from an application that performs face and pose detection of the driver through a vehicular sensor network. This application is a CNN with 11 layers (4 convolutional layers, 2 sub-sampling layers and 5 tanh layers). CNNs involve a huge amount of computations in the form of data parallel operations. The structure of CNNs is usually very regular and symmetric and allows many different permutations of operations to get to the same result. We restrict ourselves to using simple non-separable 2D convolutions, data parallel additions and tanh operations. In this context, it becomes necessary to order the operations in a convolutional layer (restricting the freedom in scheduling the operations). The CNNs used were constructed based on primitives in the torch5 library [3]. The operations involved in a single convolutional layer (with 3 input planes and 2 output planes) and its transformation are illustrated in Figure 7.

We applied our framework to two different CNNs (a small CNN with 11 layers, 1600 operators, and 2434 data structures and a large CNN with 11 layers, 7500 operators, and 11334 data structures). These CNN templates are large enough that the operator scheduling and data transfer optimizations were practically infeasible to solve optimally using the pseudo-Boolean solver. These instances were solved using the heuristics mentioned in Section 3.3.1.

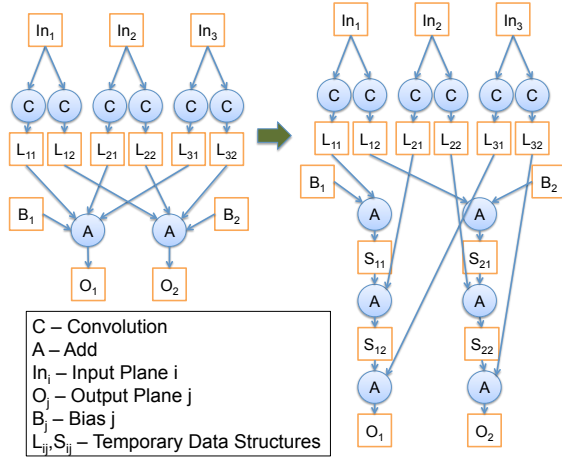


Figure 7: CNN layer transformation

They were run on both GPUs with different input image sizes ( $640 \times 480$ ,  $6400 \times 480$  and  $6400 \times 4800$ ). The baseline GPU execution times are also given in Table 2.

#### 4.2. Performance improvement and data transfer reduction

Tables 1 and 2 present the results of optimizing for data transfers between the CPU and GPU for the two different systems. From the results, it is clear that our framework helps in reducing the amount of communication between the host and GPU and that this results in superior performance. Entries in the table with “N/A” indicate infeasible configurations (data exceeds GPU memory) or inconsistent results (due to thrashing).

The final 2 entries in Table 2 gave inconsistent results. The large CNN running on GeForce 8800 GTX gives very erratic timing results. The amount of CPU-GPU memory transferred under the optimization is close to the amount of main memory (8 GB) in this case. It turns out that a significant amount of this data is active on the CPU and this leads to thrashing effects in main memory, thereby making the execution time depend heavily on OS effects (like paging and swapping). This was verified by looking at the times actually spent inside the GPU device driver using the CUDA profiler. Execution using our framework spends 51.33 seconds in the GPU driver (13.40 seconds in memcopy), whereas without our framework it spends 80.20 seconds (52.92 seconds in memcopy). The rest of the time is spent on the CPU.

#### 4.3. Scalability

For scalability analysis, we performed experiments using the edge detection template. Figure 8 shows the plot of the execution time versus the size of the input image on

the Tesla C870 platform. The edge template uses  $16 \times 16$  kernels. We define the following configuration as the “best possible” - Assume that the GPU has infinite memory and all the operations can be combined into a single optimized GPU kernel call. For the edge template, this corresponds to a single GPU kernel that takes in the input image and produces the output edge map directly. This is the optimal implementation in terms of data transfers (only input and output need to be transferred) and GPU call overhead (only one GPU kernel call). From the figure, it is clear that our methodology provides scalability and produces results that are within 20% of the best possible. Note that the baseline stops working (due to insufficient GPU memory) before the input dimension reaches 8000.

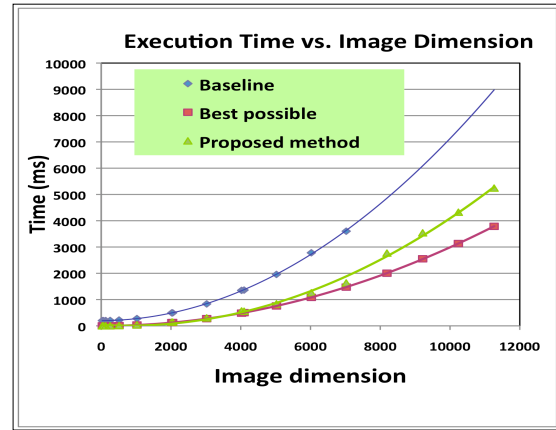


Figure 8: Performance of the edge detection template for scaling input data size

As we can see, using our methodology gives excellent speedups (1.7X to 7.8X) compared to baseline GPU implementations. With GPUs being increasingly used in general purpose computations, the need for software frameworks to hide their programming complexity has increased. Frameworks like ours will be needed in the future to help manage the complexity of heterogeneous computing platforms.

## 5. Related Work

A significant body of work has addressed the development of GPU programming frameworks and tools. The framework that is closest to ours conceptually is Accelerator [20]. Accelerator uses a operation graph representation of programs to map them to GPUs. However, the concerns of Accelerator are very different - it assumes that GPUs are not general-purpose, generates code in shader language, and tries to merge operations aggressively as it assumes that the

Table 1: Results - reduction in data transfer between the host and GPU memory

Template	Input data size	Total temporary data needed (floats)	Number of floats transferred between CPU and GPU			
			I/O transfers only (lower bound)	Baseline implementation	Optimized for Tesla C870	Optimized for GeForce 8800 GTX
Edge detection	1000x1000	6,000,512	2,000,512	13,000,512	2,000,512	2,000,512
Edge detection	10000x10000	600,000,512	200,000,512	N/A	400,000,512	400,000,512
Small CNN	640x480	59,308,709	4,870,082	157,022,568	4,870,082	4,870,082
Small CNN	6400x480	606,855,749	49,230,722	1,596,371,688	49,230,722	49,230,722
Small CNN	6400x4800	6,261,866,429	501,282,002	16,326,219,528	501,282,002	2,536,173,770
Large CNN	640x480	163,093,609	6,649,882	313,105,568	6,649,882	6,649,882
Large CNN	6400x480	1,686,960,649	67,282,522	3,212,182,688	67,282,522	67,282,522
Large CNN	6400x4800	17,664,611,329	691,377,802	33,262,586,528	760,262,830	7,877,915,800

Table 2: Results - Improvements in execution time

Template	Input data size	Time (seconds)			
		Tesla C870		GeForce 8800 GTX	
		Baseline implementation	Optimized implementation	Baseline implementation	Optimized implementation
Edge detection	1000x1000	0.28	0.036	0.19	0.034
Edge detection	10000x10000	N/A	4.12	N/A	3.92
Small CNN	640x480	1.70	0.62	1.21	0.41
Small CNN	6400x480	6.96	2.06	5.95	1.76
Small CNN	6400x4800	54.00	16.66	47.76	20.95
Large CNN	640x480	4.29	2.57	2.94	1.60
Large CNN	6400x480	15.71	6.62	13.96	5.48
Large CNN	6400x4800	262.45	112.99	N/A	N/A

overhead for a GPU call is unacceptably high. Our framework, on the other hand, does not try to generate low-level GPU code (instead relying on frameworks such as CUDA). We focus on executing computations that do not fit into the GPU memory, and in managing the CPU-GPU memory transfers efficiently and in a scalable manner.

Recently, work has been done to enable high-level frameworks like Map Reduce on GPUs [14, 6, 11, 1]. There have also been efforts to optimize code for GPUs using source-to-source compilation and auto-tuners [19, 18, 13, 16]. Streaming programming frameworks have been popular for GPUs and several of them have been proposed, notably BrookGPU [5], Peakstream (acquired by Google), and RapidMind [17]. Our framework differs in that it targets applications that do not necessarily fit into the streaming model of computation. For domain experts, we believe that providing APIs that implement domain-specific templates is the most effective means to ease the burden of GPU programming. Also, none of the streaming frameworks address the issue of mapping computations to the GPU when the data sizes are too large to fit the GPU memory.

Improving GPU programmability by presenting a unified memory space through hardware techniques has also been proposed. EXOCHI [21] is an attempt to create a unified view by trying to manage both CPU and GPU resources dynamically. CUBA [10] is an attempt to avoid data management between CPU and GPU by letting the GPU access data present in CPU memory directly and allowing it to cache them in GPU memory, thereby providing better programmability through hardware modifications.

Our work can also be viewed as an instance of understanding and exploiting the memory hierarchy present in current systems. In that respect, our work is comparable to Sequoia [12]. Sequoia is a programming language that directly deals with the memory hierarchy problem that we are trying to address. By invoking “tasks” that execute entirely in one level of memory hierarchy, Sequoia tries to optimize programs. However, Sequoia does not provide the mechanisms for reducing the data movement, which is one of our main objectives in this work.

Our framework is complementary to these efforts since we address the problem of organizing computations that do

not fit into GPU memory such that data transfer between the host and GPU is minimized.

## 6. Conclusion

We have proposed a framework for efficient and scalable execution of domain-specific templates on GPU platforms. Our framework addresses the gap in abstraction between domain experts and current GPU programming frameworks. We address the problem of executing templates whose memory footprint exceeds the available GPU memory. Our framework automatically generates an execution plan that achieves high performance by minimizing the overheads of data transfer to and from the GPU memory. We demonstrate the application of the framework to templates from the recognition domain, namely edge detection in images and convolutional neural networks. The framework can easily take advantage of new GPU platforms and advances in GPU technology. We believe that frameworks such as the one proposed here are essential to realize the potential of GPU computing by making it accessible to a broader range of programmers.

## Acknowledgements

We would like to thank Eric Cosatto and Christopher Malon for providing us with code and datasets for the cancer detection application and Iain Melvin for help with the CNN application.

## References

- [1] CUDA Data Parallel Primitives Library. <http://www.gpgpu.org/developer/cudpp>.
- [2] GPGPU community website. <http://www.gpgpu.org>.
- [3] Torch5 library. <http://torch5.sourceforge.net>.
- [4] Advanced Micro Devices, Inc. AMD Stream Computing SDK. <http://ati.amd.com/technology/streamcomputing/index.html>.
- [5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, 2004.
- [6] B. Catanzaro, N. Sundaram, and K. Keutzer. A map reduce framework for programming graphics processors. In *Proc. Third Workshop on Software Tools for Multi Core Systems (STMCS)*, April 2008.
- [7] E. Cosatto, M. Miller, H. P. Graf, and J. S. Meyer. Grading nuclear pleomorphism on histological micrographs. In *Proc. Int. Conf. Pattern Recognition*, pages 1–4, 2008.
- [8] P. Dubey. A Platform 2015 Workload Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera, 2007. <ftp://download.intel.com/technology/computing/archinnov/platform2015/download/RMS.pdf>.
- [9] N. Een and N. Sorensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, Jan 2006.
- [10] I. Gelado, J. Kelm, S. Ryoo, S. Lumetta, N. Navarro, and W. mei Hwu. CUBA: An architecture for efficient CPU/co-processor data communication. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, Jun 2008.
- [11] B. He, W. Fang, Q. Luo, N. K. Govindarajulu, and T. Wang. Mars : A mapreduce framework for graphics processors. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, October 2008.
- [12] T. J. Knight, J. Young, Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, March 2007.
- [13] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'09)*. ACM Press, Feb. 2009.
- [14] M. Linderman, J. Collins, H. Wang, and T. Meng. Merge: A programming model for heterogeneous multi-core systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, Mar 2008.
- [15] NVIDIA Corporation. NVIDIA CUDA, 2007. <http://nvidia.com/cuda>.
- [16] J. Ramanujam. Toward automatic parallelization and auto-tuning of affine kernels for GPUs. In *Workshop on Automatic Tuning for Petascale Systems*, July 2008.
- [17] RapidMind, Inc. Rapidmind Multi-core Development Platform. <http://www.rapidmind.net>.
- [18] 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 *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.
- [19] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. mei W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM.
- [20] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, Oct 2006.
- [21] P. Wang, J. Collins, G. Chinya, H. Jiang, X. Tian, M. Girkar, N. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, Jun 2007.