Synergistic Execution of Stream Programs on Multicores with Accelerators

Abhishek Udupa[†] R. Govindarajan[†‡] Matthew J. Thazhuthaveetil[†‡]

[†] Department of Computer Science and Automation
[‡] Supercomputer Education and Research Centre
Indian Institute of Science
{udupa, govind, mjt}@csa.iisc.ernet.in

Abstract

The StreamIt programming model has been proposed to exploit parallelism in streaming applications on general purpose multicore architectures. The StreamIt graphs describe task, data and pipeline parallelism which can be exploited on accelerators such as Graphics Processing Units (GPUs) or CellBE which support abundant parallelism in hardware.

In this paper, we describe a novel method to orchestrate the execution of a StreamIt program on a multicore platform equipped with an accelerator. The proposed approach identifies, using profiling, the relative benefits of executing a task on the superscalar CPU cores and the accelerator. We formulate the problem of partitioning the work between the CPU cores and the GPU, taking into account the latencies for data transfers and the required buffer layout transformations associated with the partitioning, as an integrated Integer Linear Program (ILP) which can then be solved by an ILP solver. We also propose an efficient heuristic algorithm for the work partitioning between the CPU and the GPU, which provides solutions which are within 9.05% of the optimal solution on an average across the benchmark suite. The partitioned tasks are then software pipelined to execute on the multiple CPU cores and the Streaming Multiprocessors (SMs) of the GPU. The software pipelining algorithm orchestrates the execution between CPU cores and the GPU by emitting the code for the CPU and the GPU, and the code for the required data transfers. Our experiments on a platform with 8 CPU cores and a GeForce 8800 GTS 512 GPU show a geometric mean speedup of 6.84X with a maximum of 51.96X over a single threaded CPU execution across the StreamIt benchmarks. This is a 18.9% improvement over a partitioning strategy that maps only the filters that cannot be executed on the GPU — the filters with state that is persistent across firings — onto the CPU.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Distributed Architectures—GPUs; D.1.3 [*Programming Techniques*]: Parallel Programming, Distributed Programming; D.3.2 [*Programming Languages*]: Data-flow Languages—StreamIt; D.3.4 [*Programming Languages*]: Compilers

General Terms Algorithms, Experimentation, Languages, Performance

Keywords CUDA, GPU Programming, Software Pipelining, Stream Programming, Partitioning

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'09, June 19–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-356-3/09/06. . . $5.00

1. Introduction

An interesting development in embedded systems has been the the emergence of processors integrated with accelerator architectures. For example, the OMAP 3530 processor from Texas Instruments combines a TMS320C64x+ DSP core, an ARM Cortex A8 core and a PowerVR SGX graphics accelerator, along with accelerators for image, video and audio processing, in order to meet their performance criteria. We use the term heterogeneous architecture to refer to a computing system containing multiple processing elements with different instruction set architectures (ISAs). Systems like the CellBE [6] and GPUs, which have recently acquired general purpose compute capabilities, when configured with general purpose multicore platforms are examples of heterogeneous architectures with tremendous compute power. However, these new heterogeneous accelerator architectures are difficult to program using conventional programming models. To address this difficulty, several new programming frameworks have been proposed. The OpenCL framework [2] and the CUDA framework [1] from NVIDIA are some examples. However, these frameworks have a steep learning curve associated with them, since the programmer must organize the computation as a set of data parallel kernels.

Stream programming models such as the synchronous data flow model [19, 20], Stream Flow Graphs [11, 13, 14] and StreamIt [26] have been proposed to address the difficulty of programming involved in DSP applications. They allow a programmer to express computation as a hierarchical set of filters connected by FIFO communication channels. This allows DSP and multimedia applications to be expressed naturally and frees the programmer from orchestrating the communication between program blocks. Further, these models expose *task, data* and *pipeline* parallelism [8, 12], which can be exploited in a manner most suited for the target platform equipped with various accelerators. For example, different StreamIt backends [8, 12, 18], target the RAW [7] and CellBE accelerators. This support for target-specific optimizing backends, combined with the program parallelism exposed by the StreamIt language, allows the programmer to implement DSP algorithms in a portable fashion on various platforms, without overly sacrificing performance.

In this work, we focus on compiling StreamIt programs for multi-core processors equipped with data parallel accelerators such as a GPU. Specifically, we compile StreamIt programs for execution across CPU cores and GPUs. Prior work has focused on mapping general purpose applications *only* onto the GPU [5, 25, 27]. Brook for GPUs [5] requires the programmer to structure computations as a set of kernels that execute on the GPU, while still requiring the programmer to orchestrate the communication between the kernels. Accelerator [25] takes a different approach and defines a data structure called the *data parallel array*. All operations on these data parallel arrays are offloaded onto the GPU. This requires the programmer to organize the application using data parallel arrays. The

work by Udupa et. al. [27], focuses on compiling StreamIt applications again only onto the GPU. The framework does not utilize the CPU cores for performing computation. The framework also suffers from a lack of support for StreamIt programs with *stateful* filters (described later). Other work [18, 29] has focused on compiling StreamIt applications onto heterogeneous architectures like the CellBE, again exploiting only the SPEs.

Our work in this paper orchestrates the execution of StreamIt programs across CPU cores and accelerators, specifically GPUs. There are several challenges associated. First, the CPU cores and GPU operate on separate address spaces, requiring explicit DMA commands from the CPU to transfer data into or out of the GPU address space. Second, the communication buffers between StreamIt filters need to be laid out in a specific fashion as described in [27] in order to avoid bank conflicts and *uncoalesced* accesses on the device. While this layout enables the GPU to efficiently utilize its memory bandwidth, it causes a large number of cache misses and increased memory traffic if used as is on the CPU. Thus the buffers must be *transformed* into the required layout when they are transferred between the CPU cores and the GPU. Lastly, the work partitioning between the CPU and the GPU is complicated by the DMA and buffer transformation latencies along with the fact that the filters have non-identical execution times on the two devices. This makes the work partitioning problem unamenable to simple graph partitioning heuristics. We formulate the partitioning problem as an Integer Linear Program (ILP) formulation and also propose a heuristic partitioning algorithm which compares favorably with the optimal partitions obtained from the solution to the ILP formulation. Once the task partitioning is done, the tasks are software pipelined, inserting the necessary DMA transfers and buffer layout transformations.

The main contributions of this work are:

- We describe an ILP formulation for optimal work partitioning between the CPU cores and the GPU that takes the DMA load and latencies, buffer transformation work and latencies into account.
- We propose an efficient heuristic partitioning algorithm that compares well with the optimal solutions provided by the ILP formulation. The partitioned tasks are then software pipelined to synergistically execute on the multiple CPU cores and the SMs of the GPU.
- In the process, we extend and enhance the framework proposed in [27] to support *stateful* filters.
- We implement the above mentioned schemes on the StreamIt compiler toolchain and evaluate the performance of both the schemes on a multicore with a GPU as an accelerator, demonstrating a geometric mean speedup of 6.84X, over a set of StreamIt benchmarks, with a maximum speedup of 50.96X over a single threaded CPU execution.

The approach proposed in this paper is applicable in general to other platforms with heterogeneous cores. The rest of the paper is organized as follows: Section 2 reviews the necessary background. Section 3 describes our proposed compilation methodology for synergistic execution, presenting the details of the ILP formulation, the heuristic algorithm for the work partitioning and the code generation process. Section 4 reports the performance results obtained by our scheme. Section 5 discusses related work and Section 6 concludes.

2. Background and Motivating Example

In this section, we provide a brief overview of the NVIDIA GPU architecture and the StreamIt programming Language. We then briefly describe the buffer layout scheme for GPUs proposed in [27]. In Section 2.4, we provide a motivating example for the problem addressed in this paper.

2.1 Organization of the NVIDIA GPUs

We now describe the architecture of the GeForce 8800 series of GPUs [1]. While other NVIDIA models might differ in the amount of compute and register resources available, the basic architecture remains the same. This GPU consists of 16 streaming multiprocessors (SMs), each of which in turn consists of 8 scalar units (SUs). Within an SM, the scalar units execute instructions in a lock-step fashion. The SMs have hardware support for multithreading, and periodically switch between threads to hide the latency of loads and stores. The basic schedulable entity is the *warp*, which is a contiguous group of 32 threads. Any divergence in the execution paths followed by threads within a warp causes them to serialize and incurs a performance penalty. Threads across warps may diverge without penalty. Each SM has a partitioned register file with 8192 32-bit registers, and a 16KB shared memory which is accessible by all the SUs within the same SM and is akin to a software managed cache or scratchpad memory. All SMs share a constant cache and a texture cache, which may be accessed by issuing data requests as *texture fetches* [1].

The *device* memory is connected to the SMs by a very wide (256 – 512 bits depending on the model) bus. The memory bus is capable of providing a very high bandwidth, provided all accesses by threads are *coalesced*. Essentially, a thread with *threadid* N of a warp must access an address of the form $WarpBaseAddress + N$, with $WarpBaseAddress$ suitably aligned to the size of the memory bus. With such an access pattern, each thread accesses successive memory locations and these accesses can be satisfied by a single wide (coalesced) memory access.

From a programmer's point of view, each SM executes a *block* of threads, which can consist of up to 512 threads. A GPU *kernel* call, which is dispatched to the GPU, consists of multiple *blocks*, organized as a *grid*. Each block is executed by exactly one SM, but an SM could possibly execute multiple blocks simultaneously, depending on the register and shared memory requirements of the block. A *kernel* can be dispatched asynchronously, i.e., the kernel invocation returns control to the calling thread on the CPU without waiting for the GPU to complete execution of the kernel. Any DMA transfer into the GPU or out of the GPU must be initiated by the CPU thread and can also be asynchronous.

2.2 StreamIt Overview

StreamIt allows programmers to express computation as a set of filters (also called nodes) connected by communication channels [26]. This is accomplished by a hierarchical composition of the basic StreamIt constructs: *filter*, *Pipeline*, *Split-Join* and *Feedback Loop*. A node in a StreamIt graph may only access its output channels by the `push()` method, which writes an element of a specified type onto the output FIFO. Input channels may be accessed either by the `pop()` method, which removes an element and returns it, or by the `peek()` method, which only returns the element without removing it from the FIFO. The *push, pop* and *peek* rates of a filter are static and must be specified by the programmer. By definition, the *peek rate* of a filter is greater than or equal to its *pop rate*. A filter is allowed to execute subject to its *firing rule*, which ensures that at least *peek rate* elements are available on its input FIFO and enough space is available on its output FIFO to accommodate at least *push rate* elements. An execution of a filter is called a *firing* of the filter.

Filters can be *stateless* or *stateful*. Stateful filters have persistent state which may be updated at each firing and passed on to the next firing. This mandates that the firings of stateful filters are strictly serialized, whereas multiple firings of stateless filter could execute in parallel as long as the firing rule is satisfied, without affecting correctness. In other words stateful filters are not *data parallel* and thus are unamenable for execution on the GPU and can only be sensibly executed on the CPU cores. The push and pop rates of each filter in a StreamIt program can be non-unity and non-

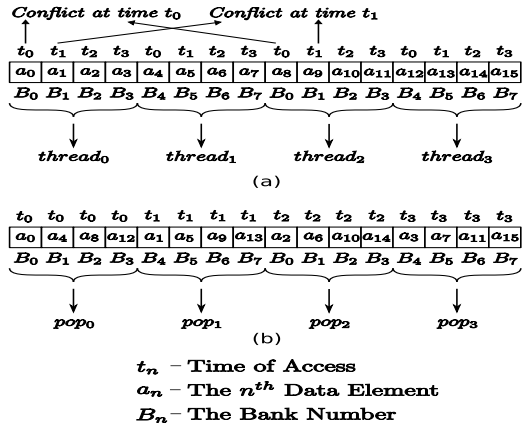

Figure 1. The buffer layout: (a) A layout which causes bank conflicts. (b) A layout which avoids all bank conflicts.

	Serial (Uncoalesced) Access (ms)	Shuffled (Coalesced) Access (ms)
CPU	14.55	187
GPU	176.6	8.1

Table 1. Execution times with serial (Uncoalesced on the GPU) and shuffled (Coalesced on the GPU) accesses

identical. A *steady state schedule* specifies the number of instances of each filter (also called the *firing rate* of the filter) that must fire in one *steady state iteration* to ensure that repetitive execution of the *steady state iteration* does not require infinite storage on any channel. That is, the production and consumption rates at each FIFO channel are balanced across a steady state schedule. The number of instances of each filter can be obtained by solving a set of equations involving the push and pop rates of the filters, called the *steady state equations* [19, 20]. One execution of a steady state schedule is termed as a *steady state iteration*.

2.3 Buffer Layout Considerations

As mentioned earlier, accesses to GPU memory must be *coalesced* in order to effectively utilize the high memory bandwidth available. In this work, we use the layout scheme proposed in [27] for the buffers on the GPU. We only briefly describe the scheme here with a simple example. Further details can be obtained from [27].

Figure 1 depicts accesses in a simplified architecture with 8 memory banks and 4 threads of execution of a filter with a pop rate of 4. As can be seen in Figure 1(a), a sequential buffer layout incurs bank conflicts and results in the conflicting accesses being serialized at every cycle. Figure 1(b) shows how all bank conflicts can be eliminated by transforming the buffer so that simultaneous accesses to device memory by the various threads never hit the same bank. The effect of this on execution time is shown in Table 1, which reports the execution time to initialize 32MB of data on the CPU and the GPU, using the serial or natural ordering of elements, which results in uncoalesced accesses and bank conflicts on the GPU, and using a *shuffled* ordering, which results in only coalesced accesses on the GPU, similar to that shown in Figure 1(b). As can be seen, uncoalesced accesses and bank conflicts cause a huge performance loss on the GPU, if a serial layout is used and the shuffled layout causes a similar performance loss on the CPU, due to cache misses arising from the staggered access pattern. This makes it important to *transform* the buffers into the form appropriate for the CPU or the GPU, depending on where they are used. We term the process of restoring a *shuffled* buffer as in Figure 1(b) into a serial ordering as *deshuffling*, and the reverse process as *shuffling*. Formally, the shuffle function, which is derived from [27] with some modifications, is defined as:

$$s(i) = b\left(\left\lfloor \frac{i}{ws} \right\rfloor + (i \bmod ws) \times \left(o \times k \times \frac{n_t}{ws}\right)\right)$$

where o, k, n_t and ws are the pop rate of the consumer on the edge under consideration, the *firing rate* of the consumer in the steady state schedule, the number of threads the consumer is executing with and the warp size of the device respectively; $s(i),\ \forall i \in [0, o \times k \times n_t)$ denotes the value at the i^{th} position in the shuffled buffer and $b(i)$ denotes the value of the i^{th} element in the natural FIFO ordering of the buffer.

Assuming that all filters on the GPU are executed with a number of threads which is a multiple of the warp size ws, the index j of the n^{th} element popped (pushed) by a thread whose *threadid* is tid in the m^{th} instance of the filter, with a pop (push) rate o is:

$$j = \text{off} + ws \times n + \left\lfloor \frac{tid}{ws} \right\rfloor \times ws \times o + (tid \bmod ws), n < o$$

where $\text{off} = m \times o \times n_t$ is the offset for the m^{th} instance of the filter. The layout scheme described in [27] shuffled the buffers at a granularity of 128 elements. However, a granularity of 32 elements is sufficient to ensure that no uncoalesced accesses occur, since the *warp size* on the device is 32 threads. We use a granularity of 32 elements here.

2.4 Motivating Example

We now provide an example that serves to highlight the need for a synergistic partitioning of StreamIt programs across the CPU cores and the GPU. Consider the simple StreamIt graph shown in Figure 2(a), which consists of a pipeline with five stages. Assume that the firing rate of each filter is unity. The execution time of each filter on the CPU and the GPU, and the latencies for transferring the required data for each channel is shown in Figure 2. For simplicity, we assume that the shuffle and deshuffle costs associated with the edges are zero, although we take them into account in the final formulation. Filter B is a stateful filter which can be executed only on the CPU. We now need to partition the remaining filters optimally to achieve the minimum possible lower bound for the Initiation Interval (MII) [21] for the software pipelined schedule. We define MII as $MII = max(RecMII, ResMII)$, where the Resource Constrained MII, ($ResMII$) and the Recurrence Constrained MII ($RecMII$) are as defined in [21]. In this work, we assume $RecMII$ to be the delay of the *heaviest* stateful filter in the stream graph. If the graph contains one or more *feedback loop* structures, then each feedback loop can easily be *fused* [8] into a stateful filter. Since stateful filters and feedback loop structures govern $RecMII$ which cannot be reduced, our approach partitions the remaining filters to execute on the CPU or the GPU such that the resulting $ResMII$ is as close to $RecMII$ as possible. Thus, the partitioning strategy tries to minimize the ResMII, by appropriately partitioning the nodes between the CPU and the GPU and modeling the DMA channel as a resource, which also contributes to $ResMII$.

Figure 2(b) shows the MII if we naïvely map filter B on the CPU and execute all the other filters on the GPU. This partition incurs DMA transfer costs from the GPU to the CPU (between filter A and B) and again from the CPU to the GPU (between filter B and C). As can be seen, a load imbalance also exists in this case, resulting in an MII of 75. Figure 2(c) shows the partitioning obtained if we try a *greedy* strategy by moving a filter to the partition (either the CPU or the GPU), where it is most beneficial to be executed. In this case, although the load has been well distributed between the CPU and the GPU, the DMA channel is over utilized, yielding an MII of 70. Finally, the partition depicted in Figure 2(d) achieves the lowest possible MII of 45 and is optimal. This example demonstrates that simple heuristics do not work well and that a more intelligent partitioning strategy that takes into account the DMA and buffer transformation latencies is called for. Also, graph partitioning heuristics

based on [17] do not help, since the weights of the nodes are not static and depend on which partition they are assigned to.

Figure 3 shows how we propose to software pipeline the program, once the optimal partitioning as in Figure 2(d) is obtained. Again, for simplicity and clarity, the shuffle and deshuffle operations that may be required are not shown.

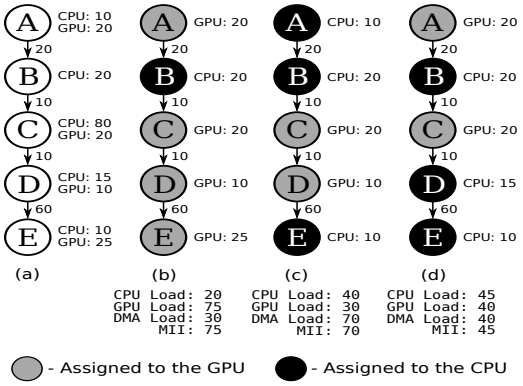

Figure 2. Motivating Example. (a) The original StreamIt graph. (b) A naïve partitioning. (c) A greedy partitioning. (d) The optimal partitioning.

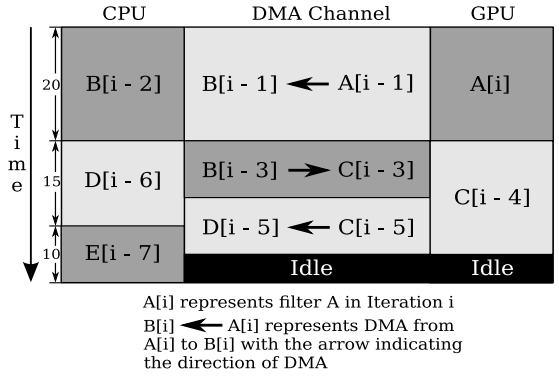

Figure 3. The Software Pipelined Kernel for the Partitioning shown in Figure 2(d). For simplicity, only one CPU core and one GPU SM is assumed.

3. Synergistic Software Pipelining

3.1 Overview of the Proposed Methodology

Figure 4 depicts the stages of our compilation methodology. First, the execution times of the filters in the StreamIt program on both the CPU and the GPU are determined by profile runs. We then use the algorithm described in [27] to determine the optimal *execution configuration*. The execution configuration defines the number of threads each filter must be executed with on the GPU (if the filter is partitioned onto the GPU). This step essentially determines the MII for all feasible combinations of execution configurations for each filter and selects the combination that yields the least MII. All filters that execute on the CPU are assumed to execute 128 times at each firing. This is done since we do not want an excessive number of CPU instances, which would in turn complicate the work partitioning. Also, 128 is a common factor for all the execution configurations we consider on the GPU, viz. 128, 256, 384 and 512 threads. Once the optimal execution configuration has been obtained, we calculate the number of instances of each filter in one steady state by solving the steady state equations. These steps are

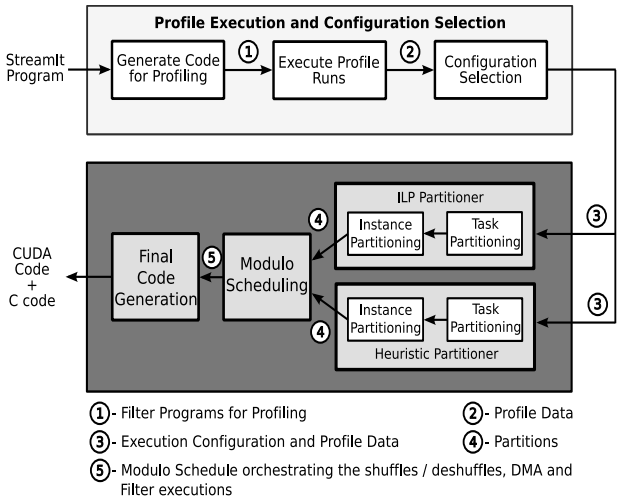

Figure 4. Overview of the compilation trajectory

the same as described in [27]. The next phase is the partitioning. We adopt a two-step approach to partitioning:

1. We first partition the *filters* themselves across the GPU and the CPU cores. This essentially partitions the stream graph into two sets, one for the GPU and the other for the CPU cores. A filter (*all* its instances) executes either on the CPU cores or on the GPU.[1] Stateful nodes and peeking nodes with large working sets (described later) are *fixed* to the CPU and are not allowed to be assigned to the GPU. We refer to this step as *filter* or *task* partitioning.
2. We then partition the *instances* of each filter across the CPU cores (if the filter is assigned to the CPU in the first step above), or across the SMs of the GPU. This helps in achieving fine grained work distribution across the CPU cores and the SMs. We refer to this step as *instance* partitioning.

If the number of instances of every filter in the StreamIt graph is small, then we may not be able to obtain good performance even with a good task partition. In this case, we *scale* up the number of instances of *all* the filters in the graph by an integral factor until there are a sufficiently large number of instances and then proceed to the instance partitioning step.

Once the partitioning phase is complete, we determine the DMA transfers and *shuffle* and *deshuffle* operations required by the partition. Table 1 shows that it is beneficial to access data in the shuffled (leading to coalesced accesses) mode in the GPU and deshuffled mode on the CPU. Hence, whenever data is transferred from the CPU to the GPU, it is first DMA'ed into the GPU and a shuffle operation is performed. For the GPU to CPU transfers, a deshuffle is performed on the GPU before the DMA transfer takes place.[2] We then orchestrate the execution of the filters, the DMA transfers and the shuffle and deshuffle operations by applying a simple modulo scheduling scheme described in Section 3.4.

The shuffle and deshuffle operations themselves are always assigned to the GPU, since it can perform these operations much faster than the CPU. We perform these operations without incurring any

[1] It is possible to consider the partitioning at the level of filter instances rather than at the level of entire filters. However, such a formulation leads to unnecessary complications, since the buffers associated with the filters must be allocated in both the CPU and GPU address spaces and data transfers must be handled at the level of filter instances. Also, since the number of instances of each filter could be in the order of a few tens to a few hundreds, the ILP formulation and the heuristic partitioner becomes unnecessarily complex. Hence we restrict our attention in this paper to partitioning only at the level of filters.

[2] There are a few exceptions to this, which we describe in Section 3.2.

bank conflicts at the device memory by staging the shuffle set into the shared memory of the SMs and incurring all bank conflicts at the shared memory. These conflicts are 1-cycle conflicts, as compared to device memory conflicts, which take 600 – 800 cycles to resolve [1], and thus do not cause significant performance losses. To ensure that all accesses to device memory are coalesced, we must shuffle (or deshuffle) at least $warpsize \times warpsize$ elements at one go, since this would mean that there are at least $warpsize$ elements that are contiguous in the serial (shuffled) ordering. Thus, we define an *instance* of a shuffle or deshuffle operation to operate on 1024 elements. We also scale the steady state execution counts of all filters to ensure that the number of elements exchanged on each buffer in one steady state iteration is a multiple of 1024. These instances form the basic schedulable unit for the shuffle and deshuffle operations. We partition these instances across the SMs of the GPU in a manner similar to partitioning the instances of filters (step 2 above). Buffers for the filters assigned to the CPU are laid out in a deshuffled fashion, while the buffers for the filters executing on the GPU are laid out in a shuffled fashion, except for the buffers for *peeking* filters, i.e., the filters that inspect tokens on their input FIFO that they do not immediately consume. These filters cannot access their buffers in a coalesced fashion, since there will always be some residual elements on their input FIFOs from a previous iteration. Thus, the input and output buffers for the peeking filters are always laid out in a sequential (deshuffled) manner. In order to avoid uncoalesced accesses at the device memory, while executing the *peeking* filters we stage the *entire* working set into shared memory before beginning execution of such filters. The output is also produced in a deshuffled fashion and is staged into shared memory to avoid uncoalesced writes. After the peeking filter has completed execution, we move the entire output set into device memory using a series of coalesced writes. This approach works well for peeking filters with small working sets. If the working set of a peeking filter is too large to be accommodated in the 16KB of shared memory available, we fix the filter to execute on the CPU.

3.2 ILP Formulation

We begin the description of the ILP formulation by defining the variables used.[3] V is the set of all nodes in the stream graph and E is the set of all directed edges in the stream graph. $C_{fixed} \subseteq V$ is the set of all filters *fixed* to the CPU. This includes all *stateful* filters and *peeking* filters whose working sets are larger than the size of the shared memory on the GPU. We use 0-1 constants $PEEK_v$, $\forall v \in V$, which is set to 1, if and only if the filter v is a peeking filter. For each filter $v \in V$, gpu_v is a 0-1 variable; gpu_v is set to 1 (by the solver) if v is assigned to the GPU. The real valued variables $shuf_{u,v} \geq 0$, $\forall (u,v) \in E$ and $deshuf_{u,v} \geq 0$, $\forall (u,v) \in E$ denote the shuffle and deshuffle cost associated with a partition for each edge $(u,v) \in E$ respectively. $DC(u,v)$ and $SC(u,v)$ are functions that return the deshuffle and shuffle cost associated with an edge (u,v) respectively, which depends on the amount of data transferred on the edge (u,v), but is a constant for a given edge in the stream graph. The real valued variables $tran_{u,v} \geq 0$, $\forall (u,v) \in E$ denote the DMA transfer cost associated with a partition for each edge (u,v). $cost_{shuf} \geq 0$ is a real valued variable indicating the total shuffle and deshuffle cost associated with a partition. The real valued variable $cost_{DMA} \geq 0$ denotes the total DMA transfer cost associated with a partition. Constants $KGPU_v$ and $KCPU_v$ denote the number of instances of a filter $v \in V$, in one steady state iteration of the software pipelined schedule, when filter v is mapped onto the GPU and the CPU respectively. These are the *scaled* steady state firings to ensure efficient execution on the respective devices. N_{cpus} and N_{gpus} are constants which denote the number of CPU cores and GPU SMs available, respectively. Lastly, $D_{cpu}(v)$ and $D_{gpu}(v)$ are the delays of *one* instance of a filter v on the CPU and the GPU respectively, which are obtained through profiling. $TGC(u,v)$ and $TCG(u,v)$ refer to the DMA transfer cost associated with the edge (u,v), from the GPU to the CPU and from the CPU to the GPU respectively.

[3] In our notation, variables in the ILP formulation are represented using lower case letters and constants using upper case letters.

3.2.1 Task Partitioning

We now describe the constraints for the ILP formulation for the task partitioning of the filters between the CPU cores and the GPU. First, we ensure that all filters in the set C_{fixed} are kept on the CPU, by the following constraint:

$$gpu_v = 0,\ \forall v \in C_{fixed} \tag{1}$$

We model the DMA costs incurred by the *cut edges*, i.e., the edges with one end on the GPU and the other on the CPU, or vice versa as:

$$tran_{u,v} \geq (gpu_u - gpu_v) \times TGC(u,v),\ \forall (u,v) \in E \tag{2}$$

$$tran_{u,v} \geq (gpu_v - gpu_u) \times TCG(u,v),\ \forall (u,v) \in E \tag{3}$$

Note that *at most* one of the above two inequalities would be active as the RHS of the other would yield a negative value.[4] Also, when both u and v are scheduled on the CPU (or both on the GPU), the RHS of the inequalities 2 and 3 would be 0. The total DMA cost associated with a partition is modeled as:

$$cost_{DMA} = \sum_{(u,v) \in E} tran_{u,v} \tag{4}$$

Inequalities (2) and (3) ensure that the $tran_{u,v}$ variables are greater than or equal to the DMA costs associated with the edge (u,v). Since the objection function is to minimize the Initiation Interval (II), these values will be pushed to their lowest permissible levels.

We now model the shuffle and deshuffle costs associated with a partition. The conditions for an edge (u,v) to require a shuffle or deshuffle operation are: (i) When filter u is assigned to the GPU and v is assigned to the CPU, deshuffle if and only if filter u does not peek; this is because if filter u peeks, then it would produce its outputs in a deshuffled order already. (ii) When both filters u and v are assigned to the GPU, deshuffle if and only if filter v peeks and filter u does not; this is because there is no need to deshuffle between a pair of peeking filters connected by a producer - consumer relationship, since the output of the producer would be in a deshuffled order already, by virtue of it being a peeking filter. (iii) When filter u is assigned to the CPU and v is assigned to the GPU, shuffle if and only if filter v does not peek. The reasoning is similar to the reasoning behind condition (i). (iv) When both filters u and v are assigned to the GPU, shuffle if and only if filter u peeks and filter v does not. Again, the reasoning is similar to the reasoning behind condition (ii). Based on these conditions, we model the shuffle and the deshuffle costs as follows. Specifically, inequalities (5) and (6) model the deshuffle costs according to the conditions (i) and (ii), while inequalities (7) and (8) model the shuffle costs according to the shuffle conditions (iii) and (iv)

$$deshuf_{u,v} \geq (gpu_u - gpu_v) \times (1 - PEEK_u) \times DC(u,v),\ \forall (u,v) \in E \tag{5}$$

$$deshuf_{u,v} \geq (gpu_u + gpu_v - 1) \times PEEK_v \times (1 - PEEK_u) \times DC(u,v),\ \forall (u,v) \in E \tag{6}$$

$$shuf_{u,v} \geq (gpu_v - gpu_u) \times (1 - PEEK_v) \times SC(u,v),\ \forall (u,v) \in E \tag{7}$$

$$shuf_{u,v} \geq (gpu_u + gpu_v - 1) \times PEEK_u \times (1 - PEEK_v) \times SC(u,v),\ \forall (u,v) \in E \tag{8}$$

[4] In this ILP formulation, it is ensured that all variables get a non-negative value (the non-negativity constraints are not shown). Thus if the RHS is negative, the constraint is subsumed by $tran_{u,v} \geq 0$.

Note that in equations (5), (6) (7) and (8), $PEEK_u$, $PEEK_v$, $SC(u,v)$ and $DC(u,v)$ are constants. The total shuffle and deshuffle costs are thus modeled by the following constraint:

$$cost_{shuf} = \sum_{(u,v)\in E} shuf_{u,v} + \sum_{(u,v)\in E} deshuf_{u,v} \quad (9)$$

We model the constraints on the Initiation Interval II as follows:

$$II \geq cost_{DMA} \quad (10)$$

$$II \geq \frac{1}{N_{gpus}} \left(\sum_{v\in V} (gpu_v \times D_{gpu}(v) \times KGPU_v) + cost_{shuf} \right) \quad (11)$$

$$II \geq \frac{1}{N_{cpus}} \sum_{v\in V} ((1 - gpu_v) \times D_{cpu}(v) \times KCPU_v \quad (12)$$

Having thus modeled the II to be the maximum of the load on the GPU, the load on the CPU and the DMA load, we can now achieve an optimal II by using an objective function which minimizes II subject to the constraints developed.

3.2.2 Instance Partitioning

We now describe the ILP formulation we use to partition the *instances* of the filters assigned to the CPU or to the GPU from the formulation in Section 3.2.1. In this phase, partitioning the instances of the filters, rather than the filters themselves, allows us to perform the partition at a much finer granularity.

We define $Nodes_{CPU}$ be the set of all nodes assigned to the CPU cores and $Nodes_{GPU}$ be the set of all nodes assigned to the GPU obtained by the filter partitioning step. We first consider the set $Nodes_{CPU}$ and define 0-1 integer variables $wcpu_{k,v,p}$, $\forall v \in Nodes_{CPU}$, $\forall k \in KCPU_v$, $\forall p \in [0, N_{cpus})$, which indicate if the k^{th} instance of a filter v is to be assigned specifically onto the processor core p. To ensure that every instance of every filter is mapped on to *exactly one* CPU core, we model the following constraint:

$$\sum_{p\in[0,N_{cpus})} wcpu_{k,v,p} = 1,\ \forall v \in Nodes_{CPU},\ \forall k \in [0, KCPU_v) \quad (13)$$

Now to ensure that no core is loaded beyond the Initiation Interval II, achieved as in Section 3.2.1, we use the following constraint:

$$\sum_{v\in Nodes_{CPU}} \sum_{k\in[0,KCPU_v)} (wcpu_{k,v,p} \times D_{cpu}(v)) \leq II,$$

$$\forall p \in [0, N_{cpus}) \quad (14)$$

By solving these constraints, a partitioning of the *instances* of the filters assigned to the CPU can be obtained which satisfies the II that has been determined. A similar formulation is used to partition the instances of the filters assigned to the GPU across the GPU SMs, as well as to partition the instances of the shuffle and deshuffle operations across the GPU SMs.

3.3 An Efficient Heuristic Algorithm

While the ILP formulation yields optimal solutions to the partitioning problem and is useful to compare other methods with a known lower bound on the II, it is not quite suited for use in a production environment, since the execution time for solving the ILP could be very large. This motivates the development of a heuristic algorithm for the task partitioning. Intuitively, we would expect the nodes assigned to the CPU to be the nodes most beneficial to execute on the CPU. We therefore define $Speedup_{cpu} = \frac{D_{gpu}(v)}{D_{cpu}(v)}$, for each filter v, where $D_{cpu}(v)$ and $D_{gpu}(v)$ are as defined in Section 3.2. A high value for this metric implies that the filter is better suited for execution on the CPU than on the GPU. Now, taking the intuition further, we would expect *not just* the nodes with the highest $Speedup_{cpu}$ values to be assigned to the CPU but also some of their neighboring nodes which could possibly have low values of $Speedup_{cpu}$, as the cost of DMA transfers and the shuffle and deshuffle costs (if necessary) would otherwise dominate and offset the benefits.

Algorithm 1 Heuristic Partitioning Algorithm

```
1:  procedure PARTITION(CPUNodes, GPUNodes)
2:      S ← Nodes sorted in decreasing order of Speedup_CPU
3:      Clusters ← GETCLUSTERS(S)
4:      bestCluster ← REFINECLUSTERS(Clusters)
5:      CPUNodes ← CPUNodes ∪ bestCluster
6:      GPUNodes ← GPUNodes − bestCluster
7:  end procedure
8:
9:  procedure GETCLUSTERS(S)                    ▷ Returns a set of clusters
10:     clusters ← ∅
11:     while ((S! = ∅) ∧ (numNodes < thresh × |V|)) do
12:         curCluster ← ∅
13:         root ← First Node in S
14:         S ← S − root
15:         curCluster ← curCluster ∪ root
16:         GROWCLUSTER(curCluster, root)
17:         clusters ← clusters ∪ curCluster
18:         numNodes ← numNodes + |curCluster|
19:         return curCluster
20:     end while
21: end procedure
22:
23: procedure GROWCLUSTER(curCluster, root)
24:     newRoot ← The node among all the predecessors of
25:     root, which is most beneficial to be added to CPUNodes
26:     in terms of reduction in II
27:     curCluster ← curCluster ∪ newRoot
28:     GROWCLUSTER(curCluster, newRoot)
29:     newRoot ← The node among all the successors of
30:     root, which is most beneficial to be added to CPUNodes
31:     in terms of reduction in II
32:     curCluster ← curCluster ∪ newRoot
33:     GROWCLUSTER(curCluster, newRoot)
34: end procedure
```

Going by this intuition, we propose the heuristic described in Algorithms 1 and 2. The procedure `partition` is the main entry point for the partitioner. The sets $CPUNodes$ and $GPUNodes$ form the initial partition with $CPUNodes = C_{fixed}$ and with $GPUNodes = V - C_{fixed}$. The first step of the partitioning is to get the set of *clusters* of nodes that are beneficial to be moved to the CPU. This is achieved by a call to the procedure `getClusters`. This procedure takes as input the set of all nodes sorted in decreasing order of their $Speedup_{cpu}$ values. It then calls the recursive procedure `growCluster` on these nodes to obtain a set of clusters. The *threshold* mentioned in line 11 determines how much of the graph is covered by the process of growing the clusters. Higher values result in a larger number of clusters. We set this parameter to 2.5 in all experiments, since it provides a reasonable compromise between speed and accuracy. Note that the clusters obtained in this fashion are not necessarily disjoint.

Growing clusters in this fashion has the limitation that it can only move contiguous regions in the stream graph to the CPU. It will not be able to capture partitions such as that shown in Figure 2(d). To overcome this limitation, we follow up the cluster growing phase with a *cluster fusion* phase. The procedure `refineClusters` performs this operation, utilizing a *smart fusion* process. The smart fusion process, described in the procedure `smartFusion`, essentially takes two clusters and greedily merges the parts of them that result in the maximum reduction in II. Thus the procedure `refineClusters` iteratively tries to apply `smartFusion` to the set of clusters until no further benefits can be obtained from the fusion. It then returns the fused cluster with the lowest II value.

The final CPU partition is thus the set of fixed CPU nodes, along with the nodes in the best cluster; while the final GPU partition

Algorithm 2 Heuristic Partitioning Algorithm (...contd.)

```
35: procedure REFINECLUSTERS(clusters)
36:     while true do
37:         tryAgain ← false
38:         srtClust ← clusters sorted in increasing
39:         order of II they achieve
40:         for i ← 0, |srtClust| do
41:             Choose a j > i such that reduction in II by calling
42:             SMARTFUSION(srtClust[i], srtClust[j])
43:             is maximum.
44:             if no such j exists then
45:                 continue
46:             else
47:                 newFusion ←
48:                 SMARTFUSION(srtClust[i], srtClust[j])
49:                 srtClust ← srtClust − srtClust[i]
50:                 srtClust ← srtClust − srtClust[j]
51:                 srtClust[i] ← newFusion
52:                 clusters ← srtClust
53:                 tryagain ← true
54:                 break
55:             end if
56:         end for
57:         if tryagain is false then
58:             return clusters[0]
59:         end if
60:     end while
61: end procedure
62:
63: procedure SMARTFUSION(clusterA, clusterB)
64:     sortedFuse ← clusterA ∪ clusterB
65:     with nodes sorted in decreasing order of Speedup_cpu
66:     bestII ← ∞
67:     fused ← ∅
68:     for i ← 0, |sortedFuse| do
69:         if II by adding sortedFuse[i] to CPU < bestII then
70:             fused ← fused ∪ sortedFuse[i]
71:         end if
72:     end for
73:     return fused
74: end procedure
```

consists of all remaining nodes in the stream graph. Note that the fixed CPU nodes are *never* considered to be moved to the GPU by the partitioning algorithm. All the II calculations in the algorithm are done by taking these fixed nodes on the CPU into account, in addition to whatever other nodes might be added. The II calculations take into account the shuffle and deshuffle costs as well as the DMA transfer costs associated with the partition.

Once the sets of CPU nodes and the GPU nodes have been decided, we use Metis [16] as a set partitioner to partition the *instances* of filters across the CPU cores or the GPU SMs. We do this by adding zero weight edges between the instances of the filters, since we are only concerned with load balancing. We also partition the instances of the shuffle and deshuffle operations using the same technique. This completes the instance partitioning step.

3.4 Modulo Scheduling

We now describe the modulo scheduling algorithm that we adopt to assign the instances of the filters to *stages* [22] in the software pipelined kernel in order to ensure correct execution. The stage assignment process essentially serves to set up the iteration differences between the various filters, while taking into account the DMA transfer latency and the shuffle and deshuffle latencies associated with the edges of the stream graph.

We use a simple modulo scheduling algorithm to orchestrate the execution of the filters, the DMA transfers and the necessary shuffles and deshuffles. Essentially, our algorithm considers nodes of the stream graph in a topologically sorted ordering. This ensures that each node is assigned a stage only after all its predecessors have been assigned stages. Our method assigns a stage number to each node based on the following rules:

1. For an edge (u, v) if u is assigned to the CPU and v is assigned to the GPU or vice versa, then $stage(v) \geq stage(u) + 2$.
2. For an edge (u, v), if both u and v are assigned to the CPU or to the GPU, then $stage(v) \geq stage(u) + 1$.
3. If an edge (u, v) requires a shuffle or a deshuffle operation, then the $stage(v)$ as computed using Rules 1 or 2 above must be further increased by one.

Rule 1 ensures that the stages of producer and consumer nodes that are across devices are separated by at least 2, to ensure that the DMA operation can be inserted in the intermediate stage. Rule 2 ensures that if the nodes are assigned to the same partition, then the stages are separated by 1, since we do not synchronize within a software pipelined kernel. Finally, Rule 3 ensures that if a shuffle or deshuffle operation is required on an edge, then the stages of the producer and consumer nodes are separated by 1 stage more than that mandated by Rules 1 and 2, so that a shuffle operation can then be inserted in the intermediate stage. The software pipelined kernel shown in Figure 3 obeys these rules.

Once the stages for the nodes have been assigned, the stages to the DMA operations and the shuffle operations are assigned. Our method assigns stages to the DMA and shuffle operations such that DMA transfers *into* the GPU occur *As Late As Possible (ALAP)*, while DMA transfers *out of* the GPU occur *As Soon As Possible (ASAP)*. This is done to conserve the limited memory space available on the GPUs, at the cost of increased memory usage in the CPU address space.

It is important to note that this algorithm will fail if there are cycles in the StreamIt graph, since no topological sort is possible in that case. While StreamIt has a *feedback loop* construct, none of the benchmarks distributed with the StreamIt toolchain utilize this construct. We assume that even if the StreamIt graph contains feedback loops, they can be *fused* [8] into a single stateful filter before the modulo scheduling algorithm is invoked.

3.5 Code Generation

The StreamIt compiler is a source to source compiler, capable of generating C-like code. We have modified the compiler to generate CUDA code for the filters assigned to the GPU and C code for the filters assigned to the CPU. The compiler also takes the buffer layout into consideration while generating code for the CPU and the GPU. The CUDA code generated by the StreamIt compiler is then compiled to native code by the *nvcc* and *gcc* compilers for the GPU and CPU respectively.

The code generation scheme we use is the *predicated kernel only* code schema described in [22]. Each SM on the GPU has access to the index of the block it is executing through a variable called `blockIdx`. We set the number of blocks to match the number of SMs on the GPU. Thus the `blockIdx` variable has a one-to-one mapping with the SMs and can be used to separate the code to be executed by each SM. Within the CPU cores, we create as many CPU *worker* threads as required at the beginning of the program execution. These threads perform the following actions indefinitely:

1. Wait for work.
2. Execute the kernel.
3. Notify completion to the main thread.

The CPU code is also predicated by the *threadid* such that each thread executes *only* the work assigned to it. Further, the threads are *pinned* to the CPU cores to ensure that the scheduler does not migrate them to another processor. Both the GPU code and the CPU code have a local variable that acts as the *staging predicate*, allowing each stage to be turned on or off as required, for the prologue and the epilogue of the software pipelined schedule.

The main thread, which executes on the CPU, is responsible for orchestrating the execution of the all the CPU threads, the DMA transfers and the GPU calls. It executes the following actions in a loop:

1. Shift the staging predicate appropriately if executing in the prologue or the epilogue phase.
2. Issue an asynchronous call to execute the GPU kernel.
3. Wake up the worker threads.
4. Issue asynchronous calls to initiate the necessary DMA transfers.
5. Wait for completion of the worker threads and the GPU kernel call.

We also perform an additional post-processing pass after the modulo scheduling to reduce the number of function calls to be executed on the GPU and the CPU. Specifically, if an SM in the GPU (or a core on the CPU) is to execute n contiguous instances $m, m+1, \ldots, m+n-1$ of a given filter v, then we compile it as a single call to the work function of v, rather than n calls. This has been done because the *nvcc* compiler that we use to compile the CUDA code for the GPUs takes time that is super-linear in the length of the GPU kernel code. Thus, reducing the number of calls to the filter work functions on the GPU, greatly reduces the compile time and memory requirements of the *nvcc* compiler. Using this technique, we were able to compile within a few minutes, programs that had previously caused the compiler to run out of memory after trying to compile for a few hours.

4. Experimental Results

We have implemented both the ILP partitioner as well as the heuristic partitioner and the simple software pipelining method described in Section 3.4 in the StreamIt compiler framework version 2.1.1, publicly available from the StreamIt website [3]. The compiler emits CUDA and C code which are then compiled by the *nvcc* and *gcc* compilers for the GPU and CPU respectively. All experiments reported in this section were performed on machines with two quad core Intel Xeon E5440 processors running at 2.83 GHz with 16GB of FB-DIMM RAM and a graphics card based on the GeForce 8800 GTS 512 GPU with 512 MB of device memory. The Linux operating system, with kernel version 2.6.24 was installed on these machines. The GPUs were driven by the NVIDIA driver version 180.11 and the CUDA 1.1 toolchain was used to build the applications. Table 4 provides the details of each benchmark that was evaluated. The benchmarks were all taken from the StreamIt benchmark suite [3]. We report the speedup over a single threaded CPU execution. We define the speedup as the ratio of t_{syn} to t_{cpu}, where t_{cpu} is the time taken for a single threaded CPU execution and t_{syn} is the time taken for a synergistic execution across the GPU and the CPU cores, with both executions performing an identical amount of work. The single threaded CPU code was generated by the cluster backend of an unmodified StreamIt compiler and compiled with *gcc* with an optimization level of `-O3`.

4.1 Comparison of Heuristic Partitioner with ILP Partitioner

The II values obtained from the ILP and heuristic task partitioner are compared in Table 4.1. The partitioning was carried out assuming 4 CPU cores and 16 GPU SMs. The results indicate that the heuristic partitioner performs quite well with average an degradation of 9.05%. Further, the degradations are well within 10% for 9 out of 12 benchmarks. A few benchmarks show higher degradations, upto 22.3%. The large degradation in Bitonic-Rec is primarily due to the fact that the II is extremely small. The ChannelVocoder benchmark is an interesting case. The optimal solution in this case is one which does not involve *any* shuffle or deshuffle operations. Thus, any deviation from the optimal would incur some shuffle or deshuffle operations and would tend to increase the II. Although our heuristic takes the shuffle operations into account while calculating the II, it fails to obtain a solution which does not require any shuffle or deshuffle operations. FFT-C owes its large degradation to the fact that it is a very small application, with only 26 filters and a relatively small II. Increasing the coverage threshold, mentioned in Algorithm 1, to 5 results in an improvement for FFT-C bringing down the degradation to about 10%.

Next we report the execution times for the partitioning using the ILP formulation and the heuristic partitioner. The heuristic partitioning method took a total of 35.9 seconds for all the benchmarks. This is considerably lower than the time taken by the ILP solver, which took a total of 2379 seconds over all the benchmarks. Also, given that the heuristic partitioner has been implemented in Java (which is the language used for the StreamIt compiler), the implementation and hence the execution time could be optimized further.

Having demonstrated that the heuristic partitioner compares favorably with the ILP partitioner in terms of the II achieved, we now compare the actual runtime of the code generated by both techniques. We apply the appropriate level of *coarsening* as in [27]. A *coarsening* of n essentially executes the software pipelined kernel n times, thereby amortizing the cost of the GPU kernel launch and the cost of waking up the worker threads. The numbers reported in Figure 5 are all for the best coarsening, which may vary across benchmarks and depends on the buffer requirements of each benchmark. Figure 5 shows the results with the applications being compiled for

Benchmark	*Filters*	$\|C_{fixed}\|$	*Description*
Bitonic	58	0	Bitonic sorting network for sorting 8 integers
Bitonic-Rec	61	0	Recursive Implementation of the bitonic sorting network
ChannelVocoder	55	1	Vocoder Implementation
DCT	40	0	8x8 DCT Implementation
DES	55	0	Implementation of the DES crypto algorithm
FFT-C	26	0	Coarse Grained FFT
FFT-F	99	0	Fine Grained FFT
Filterbank	53	0	Filter bank for multirate signal processing
FMRadio	67	0	Software FM Radio with equalizer
MatrixMult	43	0	Blocked matrix multiply
MPEG2Subset	39	1	A subset of the MPEG2 Decoder
TDE	29	0	Time Delay Equalization phase from Ground Moving Target Indicator)

Table 2. Benchmarks Evaluated

Benchmark	*II (ILP)* (ns)	*II (Heur)* (ns)	*%Degrade*
Bitonic	78778	82695	4.97
Bitonic-Rec	120576	143965	19.4
ChannelVocoder	8942998	10126982	13.24
DCT	1655026	1747211	5.57
DES	426207	454630	6.67
FFT-C	330979	405003	22.37
FFT-F	428332	443251	3.48
Filterbank	729004	785793	7.79
FMRadio	207985	217004	4.34
MatrixMult	1299710	1422917	9.48
MPEG2Subset	1918754	1991250	3.78
TDE	14646894	15751827	7.54

Table 3. Performance of the Heuristic Partitioner compared to the ILP partitioner in terms of achieved II

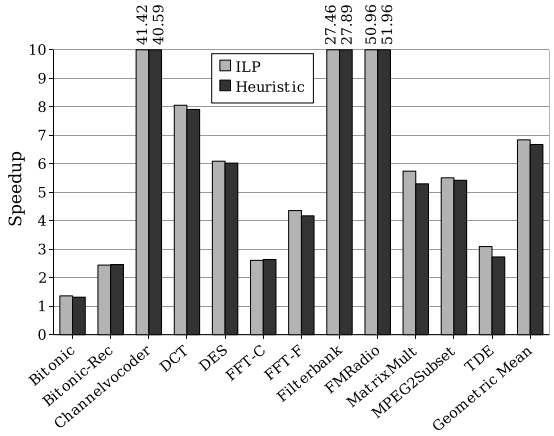

Figure 5. Performance of the ILP vs. heuristic partitioner

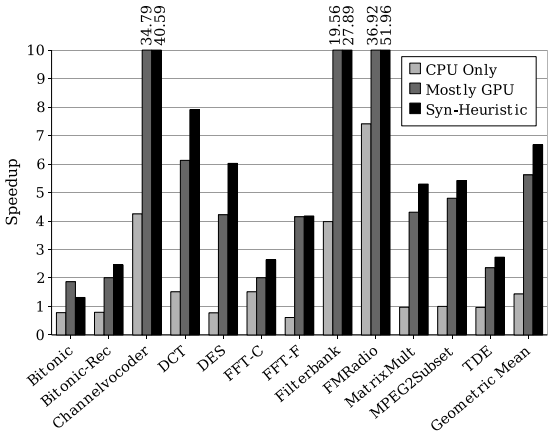

Figure 6. Comparison of Synergistic Execution with other schemes

4 CPU cores and 16 SMs. [5] Clearly, the heuristic partitioner performs quite well. It is surprising that in the case of the Bitonic-Rec, FFT-C, Filterbank and FMRadio benchmarks, the heuristic partitioner actually outperforms the ILP partitioner, albeit by a very small margin. This is despite the fact that the IIs achieved by the heuristic partitioner are larger than the IIs achieved by the ILP partitioner for these benchmarks. We attribute this to two factors. First, as demonstrated in [23], the dynamic factors which are not considered by the ILP formulation could be affecting the performance. For instance, neither the ILP nor the heuristic partitioner takes second order effects into account, such as the interaction between filters executing concurrently on the different SMs of the GPU or on the different cores of the CPU. While the profile runs try to model the contention for bandwidth by executing the same filter on all the GPU SMs or CPU cores, it is impossible to predict how the execution of another filter on a different core or SM would interact with the execution of a filter on a given core or SM. Secondly, as mentioned in Section 3.5, we perform a post-processing pass to reduce the number of device calls. The ILP partitioner tends to assign instances to the CPU cores or SMs in an arbitrary fashion, whereas the Metis partitioning tends to keep consecutive instances of filters together on the same core or SM. This results in a large number of calls being merged into a single call, when the heuristic partitioner is used, thereby eliminating the overhead of some function calls. Both these factors account for the heuristic partitioning performing slightly better than the ILP partitioner in some cases. Overall, the ILP partitioner yields a geometric mean speedup of 6.84 over the single threaded execution, while the heuristic partitioner yields a geometric mean speedup of 6.68 across the entire benchmark suite.

4.2 Comparison of the Synergistic Heuristic with Simpler Partitioning Heuristics

We now compare the performance of the synergistic partitioning approach with other approaches. Specifically, we evaluate the performance of a naïve partitioning where only the nodes that are in C_{fixed} are executed on the CPU cores, while all other nodes are executed on the GPU. We refer to this approach as "Mostly GPU". For programs without stateful filters, this is the same partitioning as done in [27]. In the second scheme, called "CPU Only", all the nodes are partitioned for execution on the CPU cores. In all cases, we report speedups relative to single threaded CPU execution.

[5] We have conducted experiments with 2 CPU cores and 6 CPU cores. The performance marginally decreases with 2 CPU cores and marginally increases with 6 CPU cores. We do not present these results here due to space limitations.

Figure 6 compares the performance of the three schemes. In most of the benchmarks, our synergistic execution performs significantly better than the CPU Only and Mostly GPU approaches, resulting in upto 51.96X speedup and a geometric mean speedup of 6.68X across the benchmark suite over a single threaded CPU execution; As opposed to 1.44X and 5.62X obtained by the CPU Only and Mostly GPU schemes respectively. Thus, the synergistic execution yields an 18.9% improvement over the Mostly GPU approach.

Next, we discuss the performance of the CPU Only scheme in detail, where our instance partitioning approach yields an average speedup of 1.44X. The Bitonic and Bitonic-Rec benchmarks perform poorly. These benchmarks are extremely bandwidth intensive. The only work that the filters in these benchmarks do is to compare and exchange values, with no other computation. Thus, these benchmarks will hit the bandwidth limitations with any scheme. Other benchmarks, notably DES and FFT-F, show a performance *degradation* (speedups less than 1) when compiled for multiple CPUs. We attribute this to the increased number of cache misses due to the partitioning being done across the CPU cores in a cache oblivious fashion. The baseline — the single threaded CPU execution — executes all the filters in a single appearance schedule [19] on the same core. Thus the producer-consumer locality results in a large degree of cache reuse, which leads to better performance in a single threaded CPU execution. Our partitioning scheme partitions the instances of the filters without considering where the producer instances of the filter are scheduled. Benchmarks like MatMul, MPEG2Subset and TDE display no speedup at all for the CPU Only scheme for the same reason. It has been demonstrated in [24] that cache aware optimizations on StreamIt programs can yield large performance gains. Benchmarks with peeking filters, such as Filterbank, FMRadio and ChannelVocoder, yield large speedups on the CPU. While the cache oblivious partitioning hurts the performance of these programs too, the effect is more than offset by the fact that executing them 128 times or more at one go allows for a large amount of cache reuse, owing to the peek set of these filters.

Our primary objective in this work has been to build a synergistic execution framework for executing stream programs across the GPU and the CPU cores. We have not taken cache considerations into account during the CPU partitioning phase. The cache considerations are irrelevant for the GPU, since they do not have a multilevel memory hierarchy in general. Our framework can easily be extended to cover this by considering two versions of CPU profile data for each filter — one with a cold cache and the other with a warm cache — and using the appropriate versions of the profile data to yield an optimal partition. With this, the proposed approach can also be used for executing StreamIt programs on multicores.

5. Related Work

Early work on stream graphs by Lee, et. al. [4, 19], have introduced the Synchronous Data Flow (SDF) model of computation, providing a sound theoretical framework for the stream programming model. Stream Flow Graphs have been studied by Gao, et. al. in [11]. Govindarajan, et. al. studied the software pipelining of Regular Stream Flow Graphs (RSFGs) [13] using a linear programming formulation. The framework was extended to reduce the buffer requirements of rate optimal r-periodic schedules in [14].

The StreamIt project has recently revived interest in the dataflow graph model of computing [3]. StreamIt introduces a *peek* construct that allows filters to inspect data on the input channels without consuming it. Software pipelining the execution of StreamIt graphs to target the RAW architecture [8, 12] and the Cell BE architecture [18, 29] and recently GPUs [27] has also been investigated. All these approaches target exploiting only homogeneous cores. Further, the framework proposed in [27] cannot handle StreamIt programs with stateful filters.

The problem of task partitioning has been studied in [15, 28]. However, these approaches have been designed with real-time embedded systems in mind and are not easily adaptable to our scenario of task partitioning for software pipelining. This work is also significantly different from [27], owing to the fact that the StreamIt program is partitioned across *heterogenenous* processing elements with *disjoint* address spaces.

Recent work on GPUs has suggested program optimization space pruning [9]. This method reduces the search space in execution configuration selection and optimization space considerably and could be used in place of the profiling methodology that we have adopted. Other work on GPUs have primarily focused on application performance tuning [10].

6. Conclusions and Future Work

We have presented a framework for synergistic execution of StreamIt programs on multicores with accelerators such as the GPU. The framework takes the DMA latencies and channel capacities into account while partitioning, by modeling the DMA channel as a resource. An efficient heuristic algorithm for partitioning the StreamIt graph across the CPU cores and the GPU SMs has been proposed and has been demonstrated to provide good results as compared to optimal solutions obtained by formulating the problem as an Integer Linear Program. Our results indicate performance gains averaging 6.84X with a maximum of 51.96X over a single threaded CPU execution for the StreamIt benchmark suite.

The proposed methodology is directly applicable for all heterogeneous accelerator based architectures. An interesting area of future research would be to perform the cache aware partitioning across the CPU cores.

References

[1] *NVIDIA CUDA Programming Guide*. URL `http://www.nvidia.com/cuda`.

[2] OpenCL Overview. URL `http://www.khronos.org/developers/library/overview/opencl_overview.pdf`.

[3] StreamIt Home Page. URL `http://www.cag.lcs.mit.edu/streamit/`.

[4] S. S. Bhattacharyya and E. A. Lee. Looped Schedules for Dataflow Descriptions of Multirate Signal Processing Algorithms. *Formal Methods in System Design*, 5(3), 1994.

[5] Ian Buck et. al. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. on Graphics*, 23(3), 2004.

[6] J. A. Kahle et. al. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4-5), 2005.

[7] Michael Bedford Taylor et. al. The RAW Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2), 2002.

[8] Michael I. Gordon et. al. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS-X: Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[9] Shane Ryoo et. al. Program Optimization Space Pruning for a Multithreaded GPU. In *CGO '08: Proc. of the sixth annual IEEE/ACM Intl. Symp. on Code Generation and Optimization*, 2008.

[10] Shane Ryoo et. al. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In *PPoPP '08: Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2008.

[11] G. R. Gao, R. Govindarajan, and P. Panangaden. Well-Behaved Dataflow Programs for DSP Computation. *ICASSP-92: IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, 1992.*, 5, Mar 1992.

[12] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *ASPLOS-XII: Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[13] R. Govindarajan and Guang R. Gao. A Novel Framework for Multi-rate Scheduling in DSP Applications. In *ASAP '93: Proc. of the 1993 Intl. Conf. on Application-Specific Array Processors*, Oct 1993.

[14] R. Govindarajan, Guang R. Gao, and Palash Desai. Minimizing Memory Requirements in Rate-optimal Schedules. In *ASAP '94: Proc. of the 1994 Intl. Conf. on Application Specific Array Processors*, Aug 1994.

[15] Junwei Hou and Wayne Wolf. Process Partitioning for Distributed Embedded Systems. In *CODES '96: Proc. of the 4th Intl. Workshop on Hardware/Software Co-Design*, 1996.

[16] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48, 1998.

[17] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Tech. Journal*, 49, Feb. 1970.

[18] Manjunath Kudlur and Scott Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. In *PLDI '08: Proc. of the 2008 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2008.

[19] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. on Computers*, 36(1), 1987.

[20] E.A. Lee and D.G. Messerschmitt. Synchronous Data Flow. *Proc. of the IEEE*, 75(9), Sept. 1987.

[21] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *MICRO 27: Proc. of the 27th annual Intl. Symp. on Microarchitecture*, 1994.

[22] B. R. Rau, Michael S. Schlansker, and P. P. Tirumalai. Code Generation Schema for Modulo Scheduled Loops. In *MICRO 25: Proc. of the 25th annual Intl. Symp. on Microarchitecture*, 1992.

[23] John Ruttenberg, Guang R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. In *PLDI '96: Proc. of the ACM SIGPLAN 1996 Conf. on Programming Language Design and Implementation*, 1996.

[24] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache Aware Optimization of Stream Programs. In *LCTES '05: Proc. of the 2005 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[25] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *ASPLOS-XII: Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[26] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC '02: Proc. of the 11th Intl. Conf. on Compiler Construction*, 2002.

[27] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software Pipelined Execution of Stream Programs on GPUs. In *CGO '09: Proc. of the seventh annual IEEE/ACM Intl. Symp. on Code Generation and Optimization*, 2009.

[28] Ti-Yen Yen and Wayne Wolf. Communication Synthesis for Distributed Embedded Systems. In *ICCAD '95: Proc. of the 1995 IEEE/ACM Intl. Conf. on Computer-aided Design*, 1995.

[29] D. Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A Lightweight Streaming Layer for Multicore Execution. *SIGARCH Computer Architecture News*, 36(2), 2008.