Software Pipelined Execution of Stream Programs on GPUs

Abhishek Udupa, R. Govindarajan, Matthew J. Thazhuthaveetil
Dept. of Computer Science and Automation
Indian Institute of Science
Bangalore, India
{udupa, govind, mjt}@csa.iisc.ernet.in

Abstract—**The StreamIt programming model has been proposed to exploit parallelism in streaming applications on general purpose multi-core architectures. This model allows programmers to specify the structure of a program as a set of filters that act upon data, and a set of communication channels between them. The StreamIt graphs describe task, data and pipeline parallelism which can be exploited on modern Graphics Processing Units (GPUs), which support abundant parallelism in hardware.**

In this paper, we describe the challenges in mapping StreamIt to GPUs and propose an efficient technique to software pipeline the execution of stream programs on GPUs. We formulate this problem — both scheduling and assignment of filters to processors — as an efficient Integer Linear Program (ILP), which is then solved using ILP solvers. We also describe a novel buffer layout technique for GPUs which facilitates exploiting the high memory bandwidth available in GPUs. The proposed scheduling exploits both the scalar units in GPU, to exploit data parallelism, and multiprocessors, to exploit task and pipeline parallelism. Further it takes into consideration the synchronization and bandwidth limitations of GPUs, yielding speedups between 1.87X and 36.83X over a single threaded CPU.

Index Terms—**Software Pipelining; GPU Programming; Stream Programming**

I. INTRODUCTION

Graphics Processing Units (GPUs) have emerged from being fixed function pipelines to massively parallel Turing complete machines, capable of performing general purpose computation. The latest generation of GPUs, consisting of hundreds of stream processing units are capable of supporting thousands of concurrently executing threads, with zero-cost hardware controlled context switching between threads. For instance the ATI Radeon 4870 has 800 stream processors connected by a 256-bit wide bus to 512 MB of memory, while the GeForce 8800 GTS 512 consists of 128 stream processors [1], organized as 16 multiprocessors, with a similar memory configuration. Each of the multiprocessors in the NVIDIA GPUs can be conceptually viewed as a very wide SIMD processor. While CPUs have traditionally added hierarchies of caches to tolerate memory latency, GPUs address the problem by providing a high bandwidth processor – memory link and supporting high degrees of Simultaneous Multithreading (SMT) within the processing elements, which allows threads to perform useful computation, while other threads are waiting for data from memory. This combination of SIMD and SMT enables the devices to achieve a peak throughput of 400 GFLOPs [2].

The high performance of GPUs, however, comes at the cost of reduced flexibility and greater programming effort from the user. For instance, in the NVIDIA GPUs, threads executing on different multiprocessors can neither synchronize in an efficient manner, nor can they communicate in a reliable and consistent manner through the device memory, within a kernel invocation [1]. Also, the GPU cannot access the memory of the host system, requiring the CPU to initiate any transfer from the host memory to device memory or vice-versa. Finally, while the memory bus is capable of delivering very high bandwidth, accesses to device memory by threads executing on a multiprocessor, need to be *coalesced* in order to actually achieve the high bandwidth.

ATI and NVIDIA have proposed the CTM [3] and CUDA [4] frameworks, respectively, for developing general purpose applications targeting the GPUs. However, both of these frameworks still require the programmer to express the program as data-parallel computations, that can be executed efficiently on the GPU. Also, programming with these frameworks tie the application to the platform. Any change in the platform would require significant rework in porting the applications. Further, these frameworks provide only low-level synchronization primitives, which the programmer would need to adapt to suit the application's requirements. All these factors contribute to the steep learning curve associated with these frameworks.

On the other hand, the StreamIt programming language, proposed to express streaming applications in a platform independent manner, aims to expose task, data and pipeline parallelism in the application in a natural way [5]. The advantage of this approach is that an optimizing compiler for a target platform can exploit parallelism in the most efficient manner for each target platform, without requiring significant effort from the programmer. A StreamIt program is expressed as a hierarchical composition of simple stream structures, which may then be *flattened* [6] into a set of filters connected by FIFO channels. The StreamIt graphs form a subset of the general multi-rate computation graphs, viz. the synchronous data flow graph or the stream flow graph [7] [8]. Apart from providing a natural way to express streaming applications, this frees the programmer from orchestrating the communication between filters. Brook [9] and Accelerator [10] are two prior works which use GPUs for streaming and general purpose computations. However, these provide a single kernel execution as the base abstraction, with the programmer having to manually handle communication between kernels [9], or require that the programmer express the program using special data structures like the data parallel array [10]. These techniques still require considerable effort by the programmer in order to transform the streaming application to execute efficiently on the GPU. A recent proposal maps StreamIt onto the CellBE platform [11], whose programming model is vastly different from that of GPUs. Figure 1 illustrates the differences between various paradigms.

To efficiently harness the compute and bandwidth resources of the CPU, parallelism must be exploited at various levels:
  1) The data parallelism across threads executing on a multiprocessor.
  2) The SMT parallelism among threads on the same multiprocessor needs to be managed to provide optimum performance. Higher levels of SMT do not automatically translate to higher performance, since the number of registers in each multiprocessor is fixed. Exceeding the number of available registers, in a blind attempt to increase the level of SMT parallelism, causes spills into the longer latency device memory, which could degrade performance.
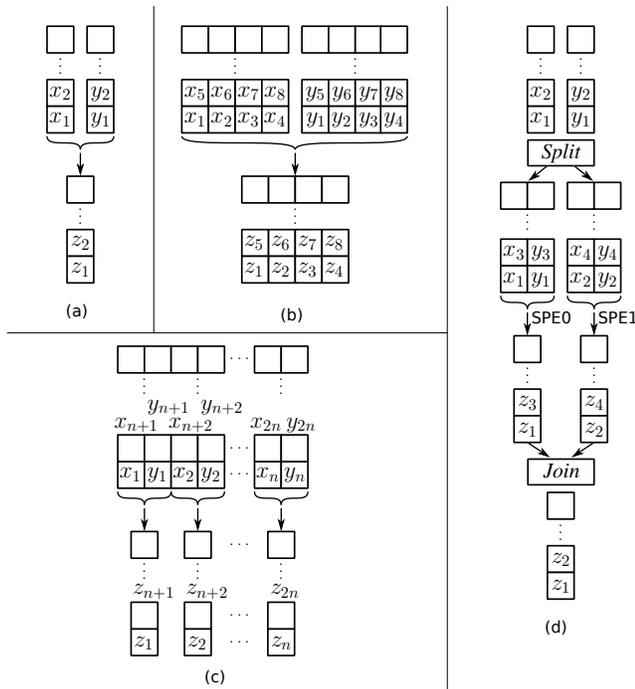
Fig. 1. Differences between various paradigms, with the execution of a *saxpy* kernel as an example. The *saxpy* kernel computes the function $z[i] = s \times x[i] + y[i]$, where s is a scalar. The vertical arrows represent threads of execution. (a) shows a unithreaded CPU, which processes elements in a loop. (b) depicts a CPU with a 4-wide SIMD unit, 4 elements are processed in one iteration, reducing the trip-count of the loop. (c) shows a GPU, where thousands of iterations of the kernel are carried out in parallel, greatly reducing the trip-count of the loop. Finally (d) describes the approach taken in [11]

3) Parallelism offered by having multiple multiprocessors on a GPU should be used to exploit the task and pipeline level parallelism in the StreamIt program.

Further, accesses to device memory need to be coalesced as far as possible to ensure optimal usage of available bandwidth. Clearly, these challenges have created a gap between being able to express streaming applications naturally and efficiently targeting the GPUs to execute these streaming applications. Our work attempts to bridge this gap.

This paper makes the following contributions:

1) We describe a software pipelining framework for efficiently mapping StreamIt programs onto GPUs.
2) We present a buffer mapping scheme for StreamIt programs to make efficient use of the memory bandwidth of the GPU.
3) We describe a profile based approach to decide on the optimal number of threads assigned to a multiprocessor, henceforth called *execution configuration* for StreamIt filters.
4) We implement our scheme in the StreamIt compiler and demonstrate a 1.87X to 36.83X speedup over a single threaded CPU on a set of streaming applications.

The rest of the paper is organized as follows: Section II provides an overview of the StreamIt language and the organization of the NVIDIA GeForce 8800 series of GPUs. Sections III and IV detail the ILP formulation used for scheduling the stream program across multiprocessors and the code generation for the GPU, respectively. In Section V we report the experimental results. Section VI discusses related work and Section VII provides concluding remarks.
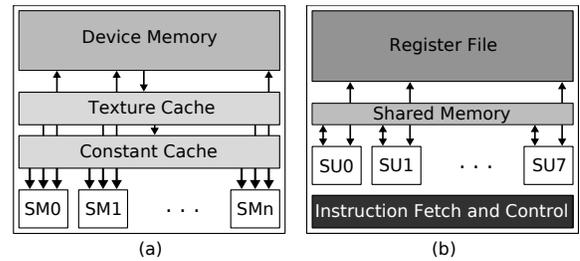

Fig. 2. Organization of the NVIDIA GeForce 8800 series of GPUs

II. BACKGROUND

In this work, we focus on mapping StreamIt programs onto NVIDIA's CUDA programming model [4]. Although the model itself is specific to NVIDIA's devices, the principles and techniques described in this work are general enough to allow stream programs to be mapped onto any programming model targeting GPUs organized as a set of streaming multiprocessors.

A. Organization of the GPU

Figure 2(a) shows the architecture of the GeForce 8800 series of GPUs [1], which consists of 16 Streaming Multiprocessors (labelled SMn in the figure), which share read-only texture and constant caches and a global device memory. Unlike a constant, a texture can be bound to any address in the device memory by the host processor prior to kernel invocation. The program executing on the GPU can access the data referred to by the texture in a cached fashion by issuing *texture fetches*. Figure 2(b) depicts the architecture of an individual SM, which consists of 8 Scalar Units (labelled SU0 – SU7 in the figure), with a common instruction fetch and thread control mechanism. The shared memory is akin to a software managed cache, accessible by all the SUs in an SM. Each SM also has a large, partitioned 32-bit register file, with 8192 registers. The memory bus connecting the SMs and the device memory is 256 – 384 bits wide (depending on the model).

Within each SM, threads execute in a *Single Instruction Multiple Data* (SIMD) fashion. The hardware periodically switches between threads to hide the latency of memory accesses. Each SM is capable of handling upto 768 threads in hardware. However, the number of threads that may actually be active simultaneously depends on the register requirements of each thread. The CUDA compiler allows the programmer to restrict the number of registers allocated per thread at compile time, If the register requirements exceed this limit, the CUDA compiler generates the necessary spill code into device memory. The basic hardware schedulable entity is the *warp*, which is a set of 32 threads with contiguous *threadIDs*. Divergence in execution paths across warps has no performance penalties, while divergence within a warp, causes the execution of the divergent threads to be serialized, incurring a performance penalty. Threads are further grouped into *thread blocks*. The maximum size of a thread block is restricted to 512 threads. All threads of a thread block are assigned to exactly one SM; the assignment being transparent to the programmer and handled by the CUDA runtime. However, multiple thread blocks may be assigned to a given SM depending on the shared memory and register requirements of each thread block. The runtime can assign a maximum of 8 thread blocks to a multiprocessor. A group of thread blocks forms a *grid*. Finally, a kernel call dispatched to the GPU through the CUDA runtime consists of exactly one grid. Thread blocks and grids may have upto three dimensions. Every
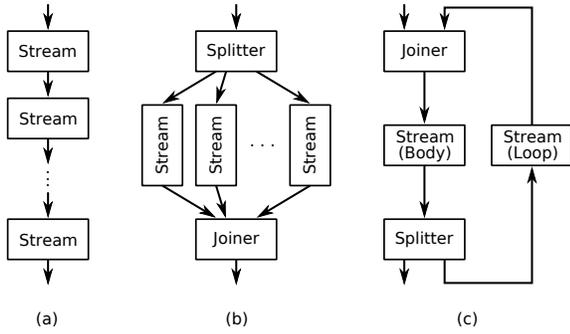
Fig. 3. The StreamIt constructs. (a) shows a *Pipeline*, (b) describes a *Split-Join* and (c) depicts a *Feedback Loop*

thread has access to its own *threadID* and the *blockID* of the thread block it belongs to by means of local variables named `threadIdx` and `blockIdx`, respectively, that are passed by the runtime. The dimensions of the block to which a thread belongs is also accessible through a variable called `blockDim`.

Efficient usage of the available memory bandwidth requires that simultaneous accesses to the device memory by the threads of a warp be to contiguous addresses, with the first warp addressing the first bank. Formally, thread N of a warp must access an address of the form $WarpBaseAddress + N$, with $WarpBaseAddress \equiv 0$ mod *Number of Banks*. Such accesses by all threads can then be *coalesced* into a single access [4].

B. The StreamIt Language

We now provide a brief overview of the StreamIt language. A StreamIt program consists of filters (also called nodes) connected by communication channels. Hierarchical composition of stream graphs is achieved using *split–join*, *pipeline* and *feedback loop* constructs [6]. Figure 3 depicts these constructs. A splitter can either be a *duplicate* splitter, — in which case it copies every element in its input FIFO to each of its outputs — or a *round robbin* splitter, in which case it copies each element from its input FIFO into exactly one output FIFO, based on its weights. For example a two way splitter with weights of $\{4, 4\}$ would copy the first four elements from its input FIFO to its first output FIFO and the next four elements to its second output FIFO. Joiners are always round robbin.

A filter may consume one or more tokens (values) from its input FIFO by executing the `pop()` method, and may produce one or more tokens into its output FIFO by executing the `push()` method. A filter may also inspect its input FIFO without consuming a token by executing the `peek(n)` method, where n is the depth into the FIFO at which the value is inspected. These are the only primitives by which filters may manipulate their input and output FIFOs. The *tokens* in the FIFO buffers can either be objects of primitive types like *int*, *float*, etc, or can be objects of user defined types. We define the *push rate* and the *pop rate* of a filter to be the number of tokens produced and consumed, respectively, during each execution of the filter. Also, the *peek depth* of a filter is defined as the depth upto which a filter can look into the input FIFO during each execution. Note that *peek depth* is always greater than or equal to *pop rate*. A filter is allowed to execute subject to its *firing rule*, which is simply that it should have at least *peek depth* tokens in its input FIFO and enough space for at least *push rate* tokens in its output FIFO.

Filters may be *stateful* or *stateless*. Stateful filters are those which have some internal state that is persistent across firings. The state

may be updated at each firing as a function of its inputs and the previous values of the state variables [12]. The outputs of stateful filters can be functions of both the values of the state variables and the inputs. On the other hand, *stateless* filters carry no state from one firing to another. The output of such filters is a function of only its inputs.

The push and pop rates of each node in a StreamIt program can be non-unity and non-identical, but are fixed and known at compile time. In order to ensure that such graphs can execute an infinite number of times with finite buffer requirements, the number of firings of different nodes can be different. The firing rate of each node can be computed by solving the *steady state rate equations* [13]. We call each firing of a particular filter an *instance* of that filter in the schedule. A schedule in which the execution counts (firing rates) of different filters are governed by the steady state equations is called a *steady state schedule*. A schedule in which the execution counts of each of the nodes are mutually prime when taken pairwise is called a *primitive steady-state schedule*. Two primitive steady-state schedules may differ in the order in which the instances of filters are invoked and hence may have different buffer requirements; from being Single Appearance Schedules (SAS) [14] [8] which require the maximum buffering, to Minimum Latency Schedules, which require the minimum [15]. A *steady state iteration* of a stream graph is defined as *one* execution of the steady state schedule.

Different instances of the same filter may be executed concurrently, as long as the firing rule is satisfied and the filter is *stateless*. Filters that have *state* inherently contain dependence edges between successive instance numbers. This implies that the state, which is updated by instance i, will be used in the computation of instance $(i + 1)$. This dependence requires a strict serial ordering of different instances of stateful filters. We consider only programs with stateless filters in this work.

III. ILP FORMULATION

In this section, we formulate the scheduling of the stream graph across the multiprocessors (SMs) of the GPU as an Integer Linear Program (ILP). We consider each instance of each filter to be the fundamental schedulable entity.

A. Definitions

We first explain the notations used in the formulation. V denotes the set of all nodes (filters) in the stream graph. N is the cardinality of V, i.e. the number of nodes in the stream graph. E denotes the set of all (directed) edges in the stream graph. k_v represents the number of firings of a filter $v \in V$, as dictated by the steady state rate equations. P is the set of all SMs in the GPU, denoted $0, 1, \ldots P_{max} - 1$. T represents the Initiation Interval (II) of the software pipelined loop. $d(v)$ is the delay or execution time of filter $v \in V$. I_{uv} and O_{uv} represent the number of elements consumed by filter v on each firing of v, and the number of filters produced by filter u on each firing of u respectively, for an edge $(u, v) \in E$. m_{uv} denotes the number of elements initially present on the edge $(u, v) \in E$.

B. Resource Constraints

We define $0 - 1$ integer variables $w_{k,v,p}$, $\forall k \in [0, k_v)$, $\forall v \in V$, $\forall p \in [0, P_{max})$, such that, $w_{k,v,p} = 1$ implies that the kth instance of the filter v has been assigned to SM p. We now model various resource constraints as follows:

$$\sum_{p=0}^{P_{max}-1} w_{k,v,p} = 1, \ \forall k \in [0, k_v), \ \forall v \in V \quad (1)$$

The above constraint ensures that each instance of each filter is assigned to *exactly one* SM.

$$\sum_{k=0}^{k_v-1} \sum_{v \in V} (w_{k,v,p} \times d(v)) \leq T, \ \forall p \in P \quad (2)$$

This constraint models the fact that all the filters assigned to an SM can be scheduled in the II specified.

C. Dependence Constraints

Before modeling the dependence constraints, we first need to ensure that the execution of any instance of any filter cannot wrap around into the next II. Consider the linear form of a software pipelined schedule [16], given by:

$$\sigma(j,k,v) = T \times j + A_{k,v}$$

where $\sigma(j,k,v)$ represents the time at which the execution of the kth instance of filter v in the jth steady state iteration of the stream graph is scheduled and $A_{k,v}$ are integer variables with $A_{k,v} \geq 0, \ \forall k \in [0, k_v), \ \forall v \in V$. We may write each $A_{k,v}$ as

$$A_{k,v} = T \times \left\lfloor \frac{A_{k,v}}{T} \right\rfloor + A_{k,v} \bmod T$$

We define

$$f_{k,v} = \left\lfloor \frac{A_{k,v}}{T} \right\rfloor \ \text{and} \ o_{k,v} = A_{k,v} \bmod T$$

Thus, the linear form of the schedule is given by:

$$\sigma(j,k,v) = T \times (j + f_{k,v}) + o_{k,v} \quad (3)$$

In the equation above, the $o_{k,v}$ indicate the time instant in the software pipelined kernel at which each filter must be scheduled to fire. And the $f_{k,v}$ serve to set up the iteration numbers of the instances of the filters, in the software pipelined schedule. $(f_{k,v} - f_{k',v'})$ denotes the number of iterations that the instance k of filter v and the instance k' of filter v' are separated by. Note that $f_{k,v}$ are integer variables greater than or equal to 0. We now constrain the starting time of filters $o_{k,v}$ as follows:

$$o_{k,v} + d(v) < T, \ \forall v \in V, \ \forall k \in [0, k_v) \quad (4)$$

This constraint ensures that *every* firing of a filter is scheduled so that it can complete within the same kernel or II. This prevents the execution of a filter from wrapping around into the next II.

We now model the dependence constraints. In order to ensure that the *firing rule* for each instance of each filter is satisfied by a schedule, the *admissibility* of a schedule is given in [13], as below:

$$\sigma(i,v) \geq \sigma\left(\left\lceil \frac{(i+1) \times I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil, u\right) + d(u),$$
$$\forall (u,v) \in E, \ \forall i \geq 0$$

where $\sigma(i,v)$ denotes the time of the ith firing of filter v. This *admissibility* condition makes an implicit assumption about the order of firing of different instances of filter v. i.e. It assumes that the firings the instances of filter v are *serialized*. Thus, a condition that ensures that the instance of a filter u, which produces the *last* quantum of data to be consumed by the *current* instance of filter v, has completed execution, before beginning execution of the current instance of filter v is sufficient. However this assumption does not hold for the model we are building, where instances of each filter *could* execute out of order, or in parallel across different SMs, as long as the *firing rule* is satisfied. So we must ensure that the schedule is *admissible* with respect to *all* the predecessors of the ith firing of the filter v. Figure
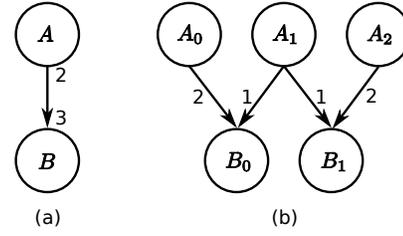

Fig. 4. (a) A multirate stream graph. Filter A pushes 2 tokens on each firing and filter B pops 3 tokens on each firing. (b) The dependency graph on each instance of each filter, with the edges indicating the number of tokens transmitted with each dependence.

4 describes this scenario. Figure 4(a) shows a multirate stream-graph where filter A pushes two elements and filter B pops three elements on each firing. Figure 4(b) shows the dependencies that exist between the various instances of filter A and B. The label represents the filter and the subscript represents the instance number, which can range from 0 to $k_v - 1$ for a given filter v. To model this behaviour, we need to analytically determine which instances of a filter u, produce data that is consumed by the ith firing of a filter v, for each edge $(u,v) \in E$. Any firing of a filter v depends on *at most* $\left\lceil \frac{I_{uv}}{O_{uv}} \right\rceil + 1$ *consecutive* firings of filter u, $\forall (u,v) \in E$. Intuitively, the ith firing of a filter v must wait for all the I_{uv} tokens produced after its $(i-1)^{th}$ firing. We model this formally as:

$$\sigma(i,v) \geq \sigma\left(\left\lceil \frac{i \times I_{uv} + l - m_{uv} - O_{uv}}{O_{uv}} \right\rceil, u\right) + d(u),$$
$$\forall l \in [1, I_{uv}], \ \forall (u,v) \in E, \ \forall i \geq 0 \quad (5)$$

Note that although it appears that there are I_{uv} constraints for each edge, there are in fact *at most* $\left\lceil \frac{I_{uv}}{O_{uv}} \right\rceil + 1$ constraints, since some constraints end up being repeated.

We now derive the dependency constraints that in turn determine the constraints on $f_{k,v}$. Consider an edge $(u,v) \in E$. The kth instance of filter v, in the jth steady state iteration is denoted by $\sigma(j,k,v)$. Since there are k_v instances of the filter v in one steady state iteration of the stream graph, using (5), we get:

$$\sigma(j,k,v) \geq \sigma\left(\left\lceil \frac{(j \times k_v + k) \times I_{uv} + l - m_{uv} - O_{uv}}{O_{uv}} \right\rceil, u\right)$$
$$+ d(u),$$

$$\forall l \in [1, I_{uv}], \ \forall j \geq 0, \ \forall k \in [0, k_v), \ \forall (u,v) \in E$$

Since $I_{uv} \times k_v = O_{uv} \times k_u$ according to the balanced rate equations for multirate stream graphs [13], we can write:

$$\sigma(j,k,v) \geq \sigma\left(\left\lceil (j \times k_u) + \frac{k \times I_{uv} + l - m_{uv} - O_{uv}}{O_{uv}} \right\rceil, u\right)$$
$$+ d(u),$$

$$\forall l \in [1, I_{uv}], \ \forall j \geq 0, \ \forall k \in [0, k_v), \ \forall (u,v) \in E$$

Simplifying further, we get:

$$\sigma(j,k,v) \geq \sigma(j'_l, k'_l, u) + d(u),$$

$$\forall l \in [1, I_{uv}], \ \forall j \geq 0, \ \forall k \in [0, k_v), \ \forall (u,v) \in E$$

Where:

$$j'_l = j + j_{lag,l}$$

$$j_{lag,l} = j + \left\lfloor \frac{1}{k_u} \left\lceil \frac{k \times I_{uv} + l - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \right\rfloor, \ \forall l \in [1, I_{uv}]$$

$$k'_l = \left\lceil \frac{k \times I_{uv} + l - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \bmod k_u, \ \forall l \in [1, I_{uv}]$$

Note that, $k \in [0, k_v) \implies j_{lag,l} \leq 0, \quad \forall l \in [1, I_{uv}]$. Now, using the form in (3), we get:

$$T \times (j + f_{k,v}) + o_{k,v} \geq T \times (j + j_{lag,l} + f_{k'_l,u}) + o_{k'_l,u} + d(u),$$
$$\forall l \in [1, I_{uv}], \ \forall j \geq 0, \ \forall k \in [0, k_v), \ \forall (u, v) \in E$$

Algebraic simplifications yield:

$$T \times (f_{k,v} - f_{k'_l,u}) \geq T \times j_{lag,l} + d(u) - (o_{k,v} - o_{k'_l,u}),$$
$$\forall l \in [1, I_{uv}], \ \forall k \in [0, k_v), \forall(u, v) \in E \qquad (6)$$

This forms a system of constraints with *at most* $\left\lceil \frac{I_{uv}}{O_{uv}} \right\rceil + 1$ constraints for each $(u, v) \in E$.

So far, we have assumed that the result of executing a filter u is available $d(u)$ time units after the filter u has started executing. However, the limitations of a GPU imply that this may not always be the case. More specifically, if the results are required from a producer instance of a filter u, which is scheduled on an SM different from the SM where the consumer instance of a filter v is scheduled, then the data produced can only be reliably accessed in the next steady state iteration. We can model this using the $w_{k,v,p}$ variables, which indicate which SM a particular instance of a given filter is assigned to. We define $0 - 1$ integer variables $g_{l,k,u,v}$, $\forall k \in [0, k_v)$, $\forall (u, v) \in E$, $\forall l \in [1, I_{uv}]$ as:

$$g_{l,k,u,v} \geq w_{k,v,p} - w_{k'_l,u,p}$$
$$g_{l,k,u,v} \geq w_{k'_l,u,p} - w_{k,v,p}$$
$$\forall k \in [0, k_v), \ \forall(u, v) \in E, \ \forall l \in [1, I_{uv}], \ \forall p \in [0, P_{max}) \qquad (7)$$

Incorporating the additional constraint into equation (6), and upon algebraic simplifications, we get two systems of constraints:

$$T \times f_{k,v} + o_{k,v} \geq T \times (j_{lag,l} + f_{k'_l,u}) + o_{k'_l,u} + d(u)$$
$$T \times f_{k,v} + o_{k,v} \geq T \times (j_{lag,l} + f_{k'_l,u} + g_{l,k,u,v})$$
$$\forall(u, v) \in E, \ \forall k \in [0, k_v), \quad \forall l \in [1, I_{uv}] \qquad (8)$$

The two constraints are set up such that the former comes into play when $g_{l,k,u,v} = 0$ and the latter comes into play when $g_{l,k,u,v} = 1$, since $o_{k'_l,u} + d(u) <= T$ at all times, from (4). The idea is that if *any* predecessor of the instance k of a filter v is scheduled on an SM different from the one where the kth instance of v is scheduled, then $g_{l,k,u,v} = 1$ for some l.

Note that (8) is independent of the iteration number j, and forms a system of *at most* $\left\lceil \frac{I_{uv}}{O_{uv}} \right\rceil + 1$ constraints for each edge $(u, v) \in E$. The inequalities (1), (2), (3), (4), (7) and (8) constitute an ILP formulation for the scheduling of the StreamIt programs on GPUs.

IV. CODE GENERATION FOR GPUS

In this section, we describe the entire process of generating a software pipelined schedule of a StreamIt program targeted for the GPU. We use the StreamIt compiler, which is a source to source compiler and is capable of generating C-like code for a variety of target platforms like the Raw architecture [5] [17] and commodity uniprocessors and multicores. This C-like code is then compiled by the native compiler for that platform. Figure 5 shows the various
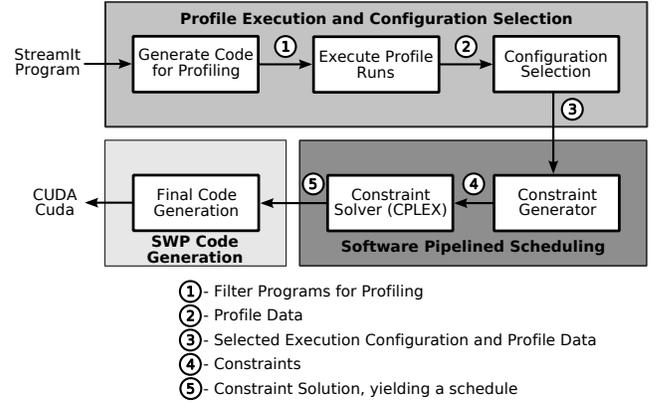

Fig. 5. Overview of the compilation process, targeting a StreamIt program onto the GPU

phases involved in our method for compiling a StreamIt application for the GPU. The following subsections describe each of these phases in detail.

A. Profiling and Execution Configuration Selection

As mentioned in sections I and II, it is important to determine the optimal execution configuration, specified by the number of threads per block and the number of registers allocated per thread. This is achieved by the profiling and execution configuration selection phase, which forms the first step of the compilation trajectory. We have modified the StreamIt compiler to generate the CUDA sources along with a driver routine for each filter in the StreamIt program. The profile code generated is compiled using the *nvcc* compiler from NVIDIA, for execution on the GPU. For each filter, we generate four versions of the profile executable. The purpose of profiling is two-fold. Firstly, the profile runs enable us to accurately estimate the execution time of each filter on the GPU, for use in the ILP formulation described in Section III. Secondly, by running multiple profile runs of the same filter, with a varying the number of threads and the number of registers the filter is compiled for, we identify the optimal number of threads — and hence the optimal level of SMT — for a given filter.

Figure 6 describes the methodology used in our profiling. We generate four versions of the executable for each filter, restricting the number of registers to 16, 20, 32 and 64 per thread. These correspond to the register requirements that will allow the kernel to run with 512, 384, 256 and 128 threads respectively, rounded up to the nearest multiple of 4. Note that if the per-thread register requirement of a kernel is less than the limit specified, it has no effect on the compilation of the kernel. Next, we execute each of these four versions with 128, 256, 384 and 512 threads, such that each execution performs the same amount of work, irrespective of the execution configuration. We accomplish this by using a parameter called *numfirings*, which represents the number of *single threaded* firings of the filter to be executed. This parameter is set to be a multiple of 128, 256, 384 and 512, and is also set to be large enough, to amortize the cost of launching the kernel on the GPU over many iterations. For the chosen execution configuration and the version of the executable, if the number of registers required per thread is greater than the available number of registers, then the kernel execution fails and the configuration in not feasible. For all the other feasible configurations we obtain the execution time on the GPU from these runs.

Fig. 6.   Algorithm for Profiling Filters

1: **for all** $i \in [0, NumFilters - 1]$ **do**
2:    **for all** $numRegs \in \{16, 20, 32, 64\}$ **do**
3:       **for all** $numThreads \in \{128, 256, 384, 512\}$ **do**
4:          execute the version of the kernel compiled for $numRegs$ registers with $numThreads$ threads for $numfirings/numThreads$ iterations.
5:          **if** kernel fails to execute due to lack of registers **then**
6:             $runTimes[i][numRegs][numThreads] \leftarrow \infty$
7:          **else**
8:             $runTimes[i][numRegs][numThreads] \leftarrow t$,
             where, t is the execution time of the kernel.
9:          **end if**
10:       **end for**
11:    **end for**
12: **end for**

Fig. 7.   Algorithm for Finding the Best Execution Configuration

1: $minII = \infty$
2: **for all** $(numRegs, numThreads) \in feasiblePairs$ **do**
3:    **for all** $i \in [0, numFilters - 1]$ **do**
4:       find $k < numThreads$ corresponding to the minimal $runTimes[i][numRegs][k]$
5:       $candidate[i] \leftarrow k$
6:    **end for**
7:    Solve the steady state equations for the execution configuration present in $candidate$ and determine the number of instances k_v for each filter v. Store it in the array $numInstances$.
8:    $curII \leftarrow 0$
9:    **for all** $i \in [0, NumFilters - 1]$ **do**
10:       $bestTime \leftarrow runTimes[i][numRegs][k]$,
          where, k is chosen as in line 4
11:       $bestTime \leftarrow bestTime \times numInstances[i]$
12:       $curII \leftarrow curII + bestTime \times (k/numfirings)$
13:    **end for**
14:    Estimate total work w done in one II by the currently chosen execution configuration. (A simple metric for this would be the number of tokens produced at the sink node of the stream graph)
15:    $curII \leftarrow curII/w$
16:    **if** $curII < minII$ **then**
17:       $minII \leftarrow curII$
18:       $bestThreads \leftarrow numThreads$
19:       $bestRegs \leftarrow numRegs$
20:       **for all** $i \in [0, NumFilters - 1]$ **do**
21:          $threads[i] \leftarrow k$, where k is chosen as in line 4
22:          $delay[i] \leftarrow runTimes[i][numRegs][threads[i]]$
23:       **end for**
24:    **end if**
25: **end for**

We choose the globally optimum execution configuration for the stream program, using the profile data. This process is described in Algorithm 7. The set *feasiblePairs* used in line 2 of Algorithm 7 consists of all pairs $(numRegs, numThreads)$, such that the configuration is a feasible configuration for all the filters. This condition is necessary, since the CUDA compiler currently does not support *extern* functions that can be called from a function executing on the device. This restriction imposes that we compile the code for *all* the filters as a single compilation unit, since calls to these filter work functions will be made by the software pipelined kernel, executing on the device. Thus we need to compile all the filters with the same register usage restrictions. Lines 3 – 7 take into account that a filter executing with $numThreads$ threads will produce and consume $numThreads$ times as many outputs and will consume $numThreads$ as many inputs as a single threaded version, and recomputes the number of instances of each filter that appears in the steady state schedule. Lines 9 – 13, calculate the candidate initiation interval (II) that will be achieved if the current configuration is chosen. The execution time is scaled in Line 12, to account for the fact that the profile run executes $(numfirings/k)$ iterations of the filter, while we require the execution time for only one iteration of the filter. Lines 14 – 15 scale the II by the amount of work done. This is necessary, since the work done by the one steady state of the various configurations may not be the same. For example, a particular configuration might yield an II of 20 time units, and produce 200 output tokens, while another might have a higher II of 40, but produce 1000 tokens. Clearly, the latter is a better alternative. The rest of the algorithm, picks the best number of threads for each filter such that the resource constrained II is minimized, and saves the optimal number of threads for each filter in an array called $threads$, and the corresponding delays for each filter in the array called $delay$

B. Software Pipelining

Note that for the optimal execution configuration selected as described in Section IV-A, the firing rates of the filters in a primitive steady state schedule are different from the corresponding firing rates in the primitive steady state schedule in the original StreamIt program. Each thread of the filter executing on the GPU corresponds to one firing in the primitive steady state schedule of the original StreamIt program. Thus the *push* and *pop* rates of the filter executing on the GPU is the base *push* rate multiplied by the number of threads chosen to execute the filter.

We generate the ILP constraints for software pipelining the stream graph for the optimal execution configurations obtained from the profiling and optimal configuration selection phase. We then use CPLEX, and industrial strength ILP solver to solve the ILP formulation. Since our ILP formulation is a constraint problem, rather than an optimization problem, the time required for solving the ILP formulation is rather low.

C. Code Generation

The solution to the ILP obtained is used to orchestrate the execution of the filters across the SMs on the GPU. Although all the SMs are required to have a single kernel entry point, the individual SMs may diverge with no performance penalty. Once the $w_{k,v,p}$, the $o_{k,v}$, and the $f_{k,v}$ have been determined from the solution of the ILP, we generate a single software pipelined kernel, with the parts required to execute on individual SMs separated using a `switch` statement. Within each SM, the instances of filters are orchestrated to execute in increasing order of the $o_{k,v}$ variables corresponding to them. Ties are broken arbitrarily, since it implies that the two instances of filters are not related by a producer–consumer relationship. We use the predicated kernel-only code generation schema described in [18], within each `case` of the `switch` statement, with the staging predicate implemented as an array, similar to the scheme described in [11].

D. Optimizing Buffer Layout

We mentioned in Section II that to achieve the high bandwidths that GPUs are capable of delivering, the accesses to device memory need
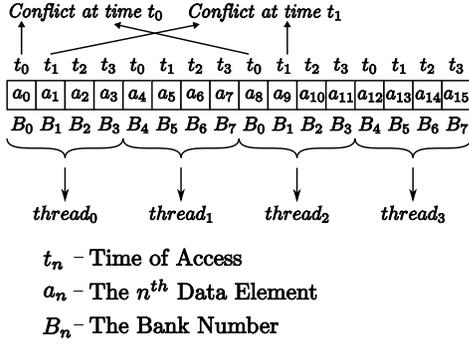
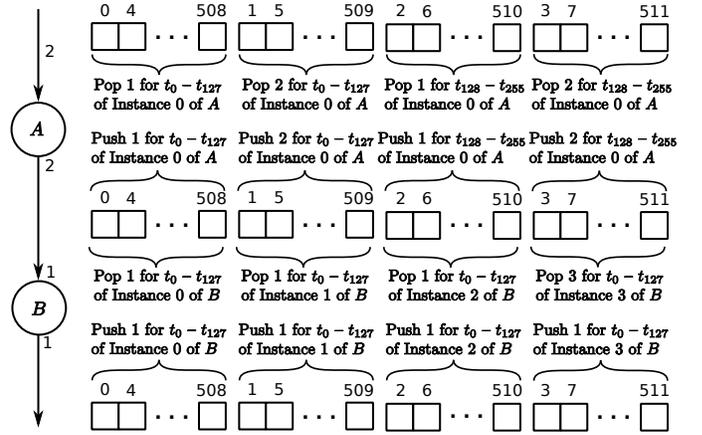Fig. 8. A sequential buffer layout, which causes bank conflicts

Fig. 9. The optimized buffer layout. The numbers at the top of each position in the buffer show the natural FIFO ordering index that the optimized buffer layout index maps to. We can see that all bank conflicts are avoided, since threads access contiguous addresses. The access pattern of each warp is exactly $WarpBaseAddress + tid$. t_n in the figure stands for threadids and push n denotes the nth push by the thread in an instance of the filter.

to be *coalesced*. We describe the problem in detail here and propose a buffer layout scheme to ensure that accesses to device memory are coalesced. Figure 8 illustrates the problem, with an example of a filter which has a *pop rate* of 4, executing with 4 threads on a device with memory organized as 8 banks. The buffer is organized sequentially, in the natural FIFO ordering of elements. Thus, $thread_0$ pops the first four elements, $thread_1$, the next four and so on. But the accesses by $thread_0$ and $thread_2$, which occur at time instant t_0, both hit bank b_0, causing these accesses to be serialized. Similar collisions occur with respect to $thread_1$ and $thread_3$. Clearly, these collisions occur at every memory access, reducing the utilization of the memory bus and degrading performance. The problem is not restricted to memory reads alone, as writes also suffer from the same conflict pattern, further degrading performance. The problem can be alleviated by streaming the entire working set of all the threads into the shared memory in each SM. The shared memory is also organized as banks, and collisions still degrade performance on shared memory, but the degradation is small, since the shared memory has an access latency of 1 cycle [1] as compared to the $400 - 600$ cycle latency of the global memory. However, this approach does not scale well for filters with large working sets. For example, a filter that pushes 64 ints and pops the same number of ints at each firing, can then be executed with a maximum of 32 threads, since each SM has only 16KB of shared memory. The low number of threads could now cause performance degradation thanks to the memory latency that has now been exposed due to low levels of SMT. Clearly, the buffer layout scheme needs to be chosen carefully to minimize bank conflict.

We leverage the fact that the *production* and *consumption* rates on the channels between filters are the same across one entire steady state in a steady state schedule, to optimize the buffer layout. We explain this with an example. Consider the stream graph shown in Figure 9. Filter A has a push and pop rate of 2 and filter B has a push and pop rate of 1. Also, filter A is determined to execute with 256 threads, while filter B is determined to execute with 128 threads. So there would be 1 instance of A in the steady state schedule and 4 instances of B. We organize the buffers such that the first 128 elements of the buffer contains the first popped elements for each of the 128 threads. This is shown in Figure 9, where we show the layout of each of the buffers in the stream graph, and what threads produce and consume into each of the positions in the buffer. We group threads into clusters of 128 threads, since this is the *gcd* of the thread block sizes that we consider. Each thread of each cluster of 128 threads, pops or pushes the first token it consumes or produces from contiguous locations in memory. In this way, it is guaranteed that no bank conflicts can occur between accesses of threads in the same

block. It is sufficient if the very first input buffer to the stream graph is shuffled, since after that, we can modify the push() and pop() routines to ensure that the data values in the buffers internal to the stream graph are communicated in a consistent manner. Formally, we define the shuffle function in terms of the number of tokens pushed or popped in one *entire* steady state execution of the stream graph as follows:

$$\text{shuffle}[i] = \text{buffer}\left[\left\lfloor \frac{i}{128} \right\rfloor + (j \bmod 128) \times steady\_pop\_rate\right],$$
$$\forall i \in [0, steady\_pop\_rate - 1] \quad (9)$$

where $steady\_pop\_rate = o_c \times k_c \times \frac{t_c}{128}$. c is the first filter of the stream graph in a topological sort ordering. o_c is the number of elements pushed by *one* thread of filter c. k_c is the number of steady state firings of c in the primitive schedule. t_c is the number of threads that c is executed with.

The index j of the nth element popped by a thread whose thread index is *tid* in an instance of a filter with pop rate o is given by:

$$j = 128 \times n + \left\lfloor \frac{tid}{128} \right\rfloor \times 128 \times o + (tid \bmod 128), \ n < o \quad (10)$$

Similarly the index k of the mth element pushed by a thread whose thread index is *tid* in an instance of a filter with push rate u is given by:

$$k = 128 \times m + \left\lfloor \frac{tid}{128} \right\rfloor \times 128 \times u + (tid \bmod 128), \ m < u \quad (11)$$

The first term in both equations sets up the offset for the push or pop. The second term serves to organize the pushes or pops into groups of 128, and the final term indicates exactly where in the current block of 128, the element needs to be pushed to or popped from.

With this buffer layout scheme, we totally avoid all bank conflicts and obviate the need to use shared memory. Further, the efficiency of the scheme is oblivious to the push and pop rates of the individual filters.

V. Experimental Evaluation

We have implemented the proposed compilation methodology on the StreamIt compiler toolchain. In this section, we demonstrate the

Benchmark	Filters	Peeking Filters	Description
Bitonic	58	0	Bitonic sorting network for sorting 8 integers.
BitonicRec	61	0	Recursive implementation of the bitonic sorting network.
DCT	40	0	8x8 Discrete Cosine Transform.
DES	55	0	Implementation of the DES encryption algorithm.
FFT	26	0	Fast Fourier Transform
Filterbank	53	16	Filter bank to perform multirate signal processing.
FMRadio	67	22	Software FM Radio with equalizer.
MatrixMult	43	0	Blocked matrix multiply.

TABLE I
BENCHMARKS EVALUATED

Bitonic	BitonicRec	DCT	DES
5308416	4472832	29360128	59768832

FFT	Filterbank	FMRadio	MatrixMult
25165824	7471104	1671168	92602368

TABLE II
THE BUFFER REQUIREMENTS FOR EACH BENCHMARK. ALL SIZES ARE IN
BYTES.

effectiveness of our methodology and compare it with two alternative approaches to solve the problem:

1) A software pipelined version that does not coalesce accesses to device memory.
2) A *serialized* execution model, where all filters execute a Single Appearance Schedule (SAS), the execution spanning across all multiprocessors available, with as much data and SMT parallelism as supported by the GPU. However, since a SAS schedule typically has the maximum buffer requirement among all schedules of a steady state solution, the buffer requirements of the SAS schedule are constrained to that of the software pipelined schedule to ensure a fair comparison between the two approaches.

A. Experimental Methodology

In order to evaluate our scheme, we use benchmarks distributed along with the StreamIt compiler toolkit version 2.1.1 [19]. The details of each benchmark and a brief description is provided in Table I. We compile each benchmark as described in Figure 5, for the CUDA framework. The *nvcc* compiler is then used to compile the resulting sources to an executable form. We choose to target 16 blocks to match the 16 multiprocessors available. I. We only consider programs with stateless filters in this study. Each benchmark was compiled and executed on a machine with dual quad-core Intel Xeon processors, running at 2.83 GHz, with 16 GB of FB-DIMM main memory. The machine runs Linux, with kernel version 2.6.18, and the NVIDIA driver version 173.14.09. We have used a GeForce 8800 GTS 512 GPU, with 512 MB of device memory for our experiments. In the results presented later, we do not report execution times, but report only the speedups relative to a single threaded CPU executing the same program. We define the speedup as:

$$speedup = \frac{t_{host}}{t_{gpu}},$$

where t_{host} is the time taken for executing same program on the host CPU mentioned, with a single thread of execution and t_{gpu} is the time taken for executing the same program on the GPU. The CPU version of the program was compiled using the uniprocessor backend of the StreamIt compiler suite. The C sources produced by the uniprocessor backend were compiled with `gcc` using optimization level `-O3`.

The timing information for the GPU version was collected using the event mechanism provided by the CUDA framework, while the timing information for the CPU version was collecting using the standard UNIX syscall `gettimeofday()`. The timing measurements

reported as speedups over the base CPU version, with both versions doing the same amount of work.

B. Experimental Results

We evaluate the following aspects of our software pipelining implementation in our experiments:

1) The effect of coarsening the granularity of the software pipelined schedule on the execution time.
2) Comparison of our methodology with a software pipelining solution that does not coalesce accesses to device memory.
3) Comparison with a serialized (one filter at a time, but fully data parallel SAS schedule, which coalesces accesses to device memory.

Figure 11 shows the effect of coarsening the granularity on the execution time. The results for each benchmark are plotted along with the geometric mean as the last bar in Figure 11 and Figure 10. In the SWPn schedule, each instance of a filter is iterated n times to increase the granularity of the GPU kernel. This does not affect the optimality of the schedule, since the delay of each filter is increased by the same proportion, thereby leaving the work distribution still uniform. This has the effect of smoothing out any anomalies in the schedule, caused by the execution times of filters being non-deterministic, due to memory access arbitration among the multiprocessors. This also has the effect of increasing the amount of work done per kernel invocation, amortizing the cost of launching kernels on the GPU over more iterations. We observe from 11 that the Optimized SWP schemes achieve a speedup of 1.87X to 36.83X for the different benchmark programs. Further the the gains start to plateau between SWP4 and SWP8 for all benchmarks.

We now compare the performance of our scheme with two other schemes. SWPNC is the same as our implementation, except that memory access coalescing is not done. For this, the profile runs are also executed without memory access coalescing. However, if the number of threads with which the filter is to be executed is such that the working set (The *push* and the *pop* set) can fit into shared memory, then we bring in the entire working set into shared memory using coalesced reads. Note that the performance gains from this approach is highly dependent on the *push* and *pop* characteristics of the filter. The *Serial* scheme is such that every filter is run as a separate kernel in a SAS schedule. We fix the number of blocks with which a filter executes to 16 — the same as the SWP scheme — and set the number of threads so that the buffer usage is less than or equal to the SWP scheme compared here, which is SWP8. No buffer sharing is performed in all our schemes; in other words, a buffer allocated for a channel is not shared with another [20]. All buffers are allocated at the beginning of the run and are not freed until the program completes execution. Table II shows the buffer requirements of the optimized software pipelined schedule which has been coarsened 8 times (SWP8) for each benchmark. Although the serial scheme executes filters in a serial fashion, we have implemented this scheme to coalesce device memory accesses.

The optimized software pipelining scheme performs better than the alternative schemes in all benchmarks, except for the MatrixMul and the DCT benchmarks, where the serial version performs slightly better. These benchmarks display a *phased* behaviour, with each phase having a splitter which exposes a large amount of task parallelism. However very little work is done in the task parallel branches before a joiner that combines the results of the task parallel branches and passes them on to the next phase. These joiners and splitters are bandwidth hungry by nature, since they only move data around, without any computation. Our scheme does not take these second order effects into account. We believe this to be skewing the actual execution times of the filters, leading to an imbalanced work distribution. The study of such phased behaviour in stream programs, is an interesting area of further research, but is beyond the scope of this work.

SWPNC performs poorly in all benchmarks, yielding a maximum speedup of only 1.22X, except in the Filterbank and FMRadio. The poor performance can be explained as being caused by non-coalesced accesses underutilizing the memory bus. The high speedups of 11.59 and 31.78 in the Filterbank and FMRadio benchmarks are due to the fact that in these benchmarks, the entire *push* and *pop* working set of all the threads fits into the shared memory for each filter and hence is brought into shared memory and written back from shared memory, using a series of coalesced accesses. Bank conflicts do occur in shared memory, however, these are conflicts are 1-cycle conflicts and do not impose a severe performance penalty, yielding a net performance gain.

We conclude the discussion on the experimental results with a mention of the efficiency of the compilation process. We have used CPLEX version 9.0, running on Linux to solve the ILP. The machine used was a quad processor Intel Xeon 3.06GHz, with 4GB of RAM. However, for all the solution times reported, CPLEX was running as a single threaded application and hence did not make use of the SMP available. The ILP solution process which is the most complex step of the compilation trajectory, is quite fast for all the benchmarks. The methodology we used to solve the ILP was to determine[1] the the lower bound on the II as $max(ResMII, RecMII)$. Once this was done, the solver was alloted 20 seconds to attempt a solution with this II. If it failed to find a solution in 20 seconds, the II is relaxed by 0.5% and the process is repeated until a feasible solution was found. It should also be noted that the compilation time is dominated by the *nvcc* compiler which takes $1 - 2$ orders of magnitude more time. All of the benchmarks took less than 30 seconds to solve, except for Bitonic, BitonicRec and DCT, which took 161, 122 and 178 seconds respectively. All solutions were found within a 5% relaxation on the II, except for FFT and FMRadio, both of which required a 7% relaxation. Thus the proposed solution is quite efficient.

VI. RELATED WORK

The scheduling of stream graphs is a fairly well studied problem in literature, with the early works by Lee, et. al [14] [8] [7], focussing on the Synchronous Data Flow (SDF) model of computation and the scheduling of such programs. Gao, et al, have defined a set of rules for constructing stream graphs [21], that restrict the general SDF model, so that the graph constructed has the property that it can be statically scheduled at compile time. Such graphs are called Regular Stream Flow Graphs (RSFG). Scheduling the execution of a stream graph at compile time is not always possible with SDF graphs, since the

[1] $RecMII$ was 0 for all the benchmarks, since none of the benchmarks in the set of benchmarks provided with the StreamIt distribution had feedback loops and we have not considered stateful filters.
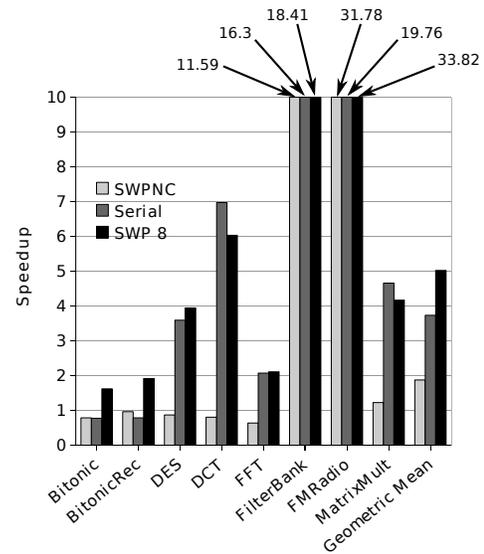

Fig. 10. SWPNC: SWP implementation with No Coalescing; Serial: Serial execution of filters using a SAS schedule.
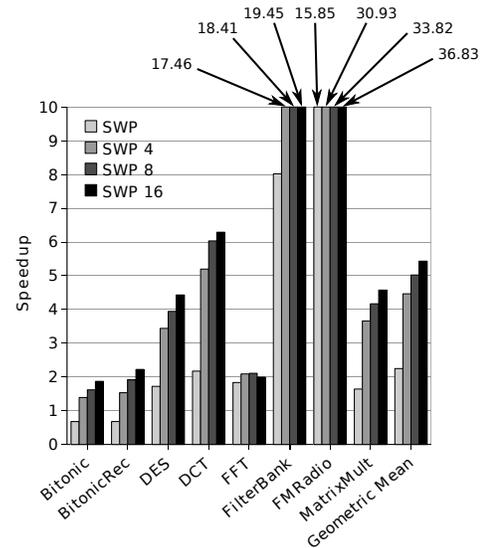

Fig. 11. Optimized SWP schedule with no coarsening; SWP 4: Coarsened 4 times; SWP 8: Coarsened 8 times; SWP 16: Coarsened 16 times.

SDF model allows for nodes to have dynamic rates. Govindarajan, et al have studied the software pipelining of RSFGs [13], using a linear programming formulation. An ILP framework for minimizing the buffer requirements of the rate optimal r-periodic schedules has also been studied [22].

More recently, the StreamIt project has revived interest in the dataflow graph model of computing. Although StreamIt introduces a *peek* construct that allows filters to inspect data on the input channels without consuming it, StreamIt graphs form a subset of RSFGs, since the *peek* can be implemented as a filter with internal state. Thus StreamIt graphs have the same schedulability properties as RSFGs. While past research has focussed on software pipelining the execution of StreamIt graphs to target the Raw architecture [23] [5] and the Cell BE architecture [11] [24], to the best of our knowledge we are the first

to have proposed a framework for compiling and executing StreamIt programs on the GPU.

Recent work on GPUs has focussed on the program optimization space pruning [2]. 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 suggested. Other work on GPUs have primarily focussed on application performance tuning [25].

VII. Conclusions and Future Work

In this paper, we describe an efficient framework for mapping StreamIt programs to GPUs. Our framework software pipelines the execution of the filters and performs both scheduling and assignment of filters to processors. Also, we present a novel buffer layout technique for GPUs which facilitates exploiting the high memory bandwidth available in GPUs. The proposed scheduling exploits both the scalar units in GPU, to exploit data parallelism, and multiprocessors, to exploit task and pipeline parallelism. Further it takes into consideration the synchronization and bandwidth limitations of GPUs, yielding speedups between 1.87X and 36.83X over a single threaded CPU. Handling stateful filters on GPUs is a possible future work.

Acknowledgment

References

[1] *NVIDIA CUDA Programming Guide.* [Online]. Available: www.nvidia.com/cuda

[2] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-M. 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*, 2008, pp. 195–204.

[3] *ATI CTM Guide.* [Online]. Available: http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

[4] NVIDIA CUDA. [Online]. Available: www.nvidia.com/cuda

[5] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs," in *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 151–162.

[6] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, 2002, pp. 179–196.

[7] E. Lee and D. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept. 1987.

[8] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 777–786, 2004.

[10] 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*, 2006, pp. 325–335.

[11] M. Kudlur and S. Mahlke, "Orchestrating the Execution of Stream Programs on Multicore Platforms," in *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 114–124.

[12] S. Agrawal, W. Thies, and S. Amarasinghe, "Optimizing Stream Programs using Linear State Space Analysis," in *CASES '05: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2005, pp. 126–136.

[13] R. Govindarajan and G. Gao, "A Novel Framework for Multi-rate Scheduling in DSP Applications," in *ASAP '93: Proceedings of the 1993 International Conference on Application-Specific Array Processors*, Oct 1993, pp. 77–88.

[14] S. S. Bhattacharyya and E. A. Lee, "Looped Schedules for Dataflow Descriptions of Multirate Signal Processing Algorithms," *Formal Methods in System Design*, vol. 5, no. 3, pp. 183–205, 1994.

[15] M. Karczmarek, W. Thies, and S. Amarasinghe, "Phased Scheduling of Stream Programs," in *LCTES '03: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, 2003, pp. 103–112.

[16] R. Govindarajan, E. R. Altman, and G. R. Gao, "Minimizing Register Requirements Under Resource-constrained Rate-optimal Software Pipelining," in *MICRO 27: Proceedings of the 27th annual International Symposium on Microarchitecture*, 1994, pp. 85–94.

[17] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The RAW Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[18] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, "Code Generation Schema for Modulo Scheduled Loops," in *MICRO 25: Proceedings of the 25th annual International Symposium on Microarchitecture*, 1992, pp. 158–169.

[19] StreamIt Home Page. [Online]. Available: http://www.cag.lcs.mit.edu/streamit/

[20] P. K. Murthy and S. S. Bhattacharyya, "Buffer Merging—A Powerful Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications," *ACM Transactions on Design and Automation of Electronic Systems*, vol. 9, no. 2, pp. 212–237, 2004.

[21] G. Gao, R. Govindarajan, and P. Panangaden, "Well-Behaved Dataflow Programs for DSP Computation," *ICASSP-92: IEEE International Conference on Acoustics, Speech, and Signal Processing, 1992.*, vol. 5, pp. 561–564 vol.5, Mar 1992.

[22] R. Govindarajan, G. Gao, and P. Desai, "Minimizing Memory Requirements in Rate-optimal Schedules," in *ASAP '94: Proceedings of the 1994 International Conference on Application Specific Array Processors*, Aug 1994, pp. 75–86.

[23] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A Stream Compiler for Communication-Exposed Architectures," in *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 291–303.

[24] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe, "A Lightweight Streaming Layer for Multicore Execution," *SIGARCH Computer Architecture News*, vol. 36, no. 2, pp. 18–27, 2008.

[25] 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*, 2008, pp. 73–82.