Efficient Compilation of Stream Programs for Heterogeneous Architectures: A Model-Checking based approach

IISc-CSA-TR-2015-2

http://www.csa.iisc.ernet.in/TR/2015/2/

Rajesh Kumar Thakur
Dept. of Computer Science and Automation
Indian Institute of Science, India
rajesh@csa.iisc.ernet.in

Y.N. Srikant
Dept. of Computer Science and Automation
Indian Institute of Science, India
srikant@csa.iisc.ernet.in

ABSTRACT

Stream programming based on the synchronous data flow (SDF) model naturally exposes data, task and pipeline parallelism. Statically scheduling stream programs for homogeneous architectures has been an area of extensive research. With graphic processing units (GPUs) now emerging as general purpose co-processors, scheduling and distribution of these stream programs onto heterogeneous architectures (having both GPUs and CPUs) provides for challenging research. Exploiting this abundant parallelism in hardware, and providing a scalable solution is a hard problem.

In this paper we describe a coarse-grained software pipelined scheduling algorithm for stream programs which statically schedules a stream graph onto heterogeneous architectures. We formulate the problem of partitioning the work between the CPU cores and the GPU as a model-checking problem. The partitioning process takes into account the costs of the required buffer layout transformations associated with the partitioning and the distribution of the stream graph. The solution trace result from the model checking provides a map for the distribution of actors across different processors/-cores. This solution is then divided into stages, and then a coarse grained software-pipelined code is generated. We use CUDA streams to map these programs synergistically onto the CPU and GPUs. We use a performance model for data transfers to determine the optimal number of CUDA streams on GPUs. Our software-pipelined schedule yields a speedup of upto 55.86X and a geometric mean speedup of 9.62X over a single threaded CPU.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Metrics—*Data-flow Languages, StreamIt*; D.3.4 [**Programming Languages**]: MetricsCompilers, Processors

General Terms

Experimentation, Languages, Algorithms, Performance

Keywords

Software Pipelining, Model Checking, Stream Programming, Partitioning, GPU Programming, CUDA

1. INTRODUCTION

Stream programs based on the synchronous data-flow model (SDF)[13] are a class of programs which are represented as graphs of independent actors interacting through a FIFO communication channel, with a requirement that the number of data items produced or consumed by each actor is known a-priori.

Stream programs expose parallelism in multiple forms. Data parallelism exists when an actor is stateless and can thus be replicated. Data parallelism provides load-balanced and abundant parallelism (as long as input data is available). A stateful actor cannot be replicated and poses a limit to the scalability of parallelization, as the work of that actor cannot be divided. The most load-intensive stateful actor becomes a bottleneck. Task and pipeline parallelism are the other forms of parallelism that stream programs also exposes. Exploiting all three forms of parallelism from stream programs poses challenges to scheduling and extraction of optimal performance from the application.

Applications such as audio, video, digital signal processing, and data analysis can be naturally expressed using stream programs; these are the applications which will be in the fore as computing moves towards data-centric applications and to the mobile and embedded space. Stream programs with explicit and regular communication are a natural fit for exploiting the coarse-grained parallelism suitable for multicore and heterogeneous (CPU and/or GPU combined) architectures. Streaming applications thus have spawned a number of streaming languages such as, StreamIt[17], Brook[4] Lime[1], etc.

The current trend in many-core architectures with heterogeneous processing capabilities with a goal to obtain maximum performance from the system has lead to interesting research challenges in managing and running programs on these machines. The heterogeneity incorporated into the system encompasses computing cores of all dimensions: CPU to handle conventional workloads, and massive number crunching GPUs for an embarrassingly parallel application. Combining these two architectures together to run in synergy needs a sophisticated compiler and a lot of programming effort and time. Current state-of-the-art approaches to map stream programs onto these architectures use integer linear programming to obtain an optimal schedule. The extremely large execution time of commercial ILP solvers for this application makes these approaches highly impractical for production compilers[14]. Model-checking has been shown to give a better performance as compared to ILP methods[14].

This paper makes the following contributions:

1. A model-checking based integrated fission and partitioning method to efficiently map actors in a StreamIt program onto a heterogeneous system that is simpler than that of [14].

2. A profile-based approach to decide the optimal number of streams on GPUs required to implement a coarse grained software-pipelined stream program.

3. An algorithm to derive a coarse grained software-pipelined schedule for StreamIt programs that uses streams to perform concurrent kernel execution on GPUs. To the best of our knowledge, this is the first effort to use this feature for stream languages.

We have implemented our scheme in the StreamIt compiler and we achieve a speedup of upto 55.86X and a geometric mean speedup of 9.62X over a single threaded CPU on a set of streaming benchmark applications.

2. RELATED WORK

Synchronous dataflow graphs have been studied extensively in the literature[13], with the focus on design of languages to express stream graphs[17], which exploit the abundant parallelism exposed by these graphs. StreamIt[17] has been proposed to address the difficulty of programming involved in DSP applications. Gordon et.al[10] exploit the parallelism available in StreamIT on the RAW architecture, where each actor runs as a separate thread and is deployed on a separate core. The inter-thread communication was managed using a buffer, and data transfers were carried out in batch mode, instead of transferring one element at a time. This implementation exploited parallelism in the stream graph exclusively with stateless filters, and stateful filters in the stream graph disabled all parallelisation.

Kudlur and Mahlke[12] built ILP formulations minimizing the initiation interval(II) (makespan) on IBM Cell platforms. Their work consists of two major phases. First, an ILP formulation is given for the integrated fission and actor assignment for the CellBE architecture, which balances the work load onto the processors. Secondly, a modulo scheduling algorithm is presented which pipelines the execution of the stream graph onto the Cell. It is not trivial to adapt this approach to an NVIDIA CPU-GPU combination and solving the ILP is expensive.

Udupa et al. [19] also formulate an ILP problem for generating a coarse-grained software-pipelined schedule for the stream graph and use a technique similar to that of Kudlur et.al.[12]. Udupa et al. [19] generate code for a combination of NVIDIA GPUs and multi-core CPU. They also propose a heuristics- based algorithm for partitioning actors onto processors that does not take the communication cost into account. They perform shuffling and deshuffling operations to coalesce all memory accesses to the GPU which translates to better execution times on heterogeneous architectures. In [18], Udupa et al. use ILP to generate a software-pipelined schedule exclusively for GPUs. However, neither of these works use the facility of *streams* to execute multiple kernels

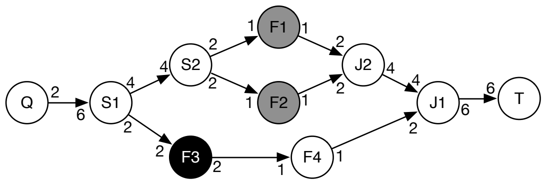


Figure 1: SDF Stream Graph

on the GPU. Our work targets architectures with both CPU and GPU and can generate code for all the combinations: exclusively CPU or GPU and CPU-GPU combination, which neither of the above works can.

Malik et. al.[14] perform an extensive quantitative evaluation comparing the heuristic solutions of several existing works on stream graphs, and propose a model-checking based approach to obtain the optimal makespan for stream graphs. They also compare the time taken to obtain the partitioning of actors onto processors with an ILP based approach and conclude that model checking provides an optimal solution taking 44% less time than ILP based approaches. They stop short of generating code and do not deal with machine related issues. Our CTL(Computation Tree Logic) model is simpler than that of Malik et. al.[14], because we do not incorporate communication into it. We produce lesser number of automata as well.

Farhad et al. [8] propose a heuristic algorithm for partitioning an SDF graph onto multicore homogeneous platforms. Carpenter et al. [5] target load-balancing filters on the architecture and their algorithm provides a suboptimal result. None of the works above can handle loops and peeking filters in a stream graph, which we do.

3. PRELIMINARIES

3.1 Stream Programming Model

A stream graph is an abstract representation of a program in the dataflow model, which is defined as $G = \{V, E\}$, where $V = \{v_1, ..., v_n\}$ is the set of actors/filters, and $E \subseteq V \times V$ is the set of FIFO communication channels between actors. A channel $(v_i, v_j) \in E$ buffers tokens (data elements) which are passed from the output of v_i to the input of v_j. Synchronous dataflow (SDF) restricts the model by fixing the number of input and output tokens of a filter v_i.

A *periodic schedule* is a finite sequence of invocations of actors, in which each actor is invoked at least once. A periodic schedule is computed at compile time and it produces no net change in the system i.e. the number of tokens on each edge of the stream graph is the same before and after executing the schedule. This state of a stream graph which can be executed ad-infinitum without any further increase in memory is called the *steady state*.

A periodic schedule is a positive integer vector $\eta \in \mathbb{N}^n$ called the *natural granularity* whose elements correspond to actors in the stream graph. Element $\eta_i, \forall i \in 1, .., n$ is equal to the minimum number of iterations of actor i in order to obtain a steady state.

We explain our approach using a simple SDF graph shown in Figure 1. The nodes in the graph are called actors (filters in StreamIt). Actor Q continuously pushes 2 units of data into the stream. The Splitter $S1$ consumes 6 units of data and splits copies for the filters $S2$ and $F3$ (4 and 2 resp.) in the stream. The data stream is subsequently sent to different processes running in parallel ($F1, F2$ with $F3$). $F1, F2$, and $F4$ are stateless filters, that is, every invocation of these filters is independent of any other previous invocation. $F3$ is a stateful filter. The end result of these parallel running actors then gets aggregated and is sent to T for further processing. The communication channels between the actors, shown as edges in the graph, are First-In First-Out (FIFO) channels. Each edge is annotated with the number of tokens produced and consumed by the connected actors.

The actors are coloured white, gray, and black to represent stateless, identity, and stateful filters . The natural granularity for the example in Figure 1 is $\eta = \{3, 1, 1, 2, 2, 1, 1, 2, 1, 1\}$ for filters {Q, S1, S2, F1, F2, J2, F3, F4, J1, T } respectively.

3.2 NVIDIA GPUs and CUDA streams

Our compilation target is a heterogeneous combination of cores with different ISA(instruction Set Architecture) and address space, including both multicore CPUs and NVIDIA GPUs. We generate pthreads for multi-core systems and CUDA for NVIDIA GPUs [7].

Broadly, all GPUs consists of n streaming multiprocessors (SMs), each of which consists of m scalar units (SUs). Within SM's, the SU's execute in a lock-step fashion. The basic schedulable entity on GPUs is the *warp*, which is a contiguous group of 32 threads, and each *warp* has it own address. A thread in a *warp* can be identified by its *threadid* and the memory it accesses is at an offset from the *warp's* base address. Sets of such *warp* form a *block*. A GPU *kernel* consists of multiple *blocks*, organised as *grids*. Each *block* is executed on exactly one SM. We omit the details of GPUs, as we are not restricted by a specific GPU architecture and our approach is portable to any of system having NVIDIA GPUs.

In order to overlap computation and communication, CUDA permits execution of programs in several stages, called *streams* [9]. CUDA defines a stream as a sequence of operations that are performed in order on the device. Typically, such a sequence contains one memory copy from host to device, which transfers input data; one kernel launch, which uses this input data; and one memory copy from device to host, which transfers results. Also, streams let the programmers launch multiple kernels onto GPUs, which facilitates execution of multiple actors onto GPUs in parallel. To the best of our knowledge, ours is the first effort to use streams in a StreamIT compiler.

3.3 Communication Overhead

A stream graph clearly establishes the precedence between producer and consumer actors, and thus needs this dependency to be preserved during the execution of stream programs. When a producer actor and the consumer actor are mapped onto different processors, the data has to be communicated to the consumer. I our implementation on the heterogeneous architecture, actors are mapped onto disjoint address spaces, which consist of both CPU and GPU memory space. The communication of data should thus be through an explicit DMA transfer. If these transfers are not avoided, or not carefully overlapped with useful work, the communication overhead will dominate the execution times of stream programs.

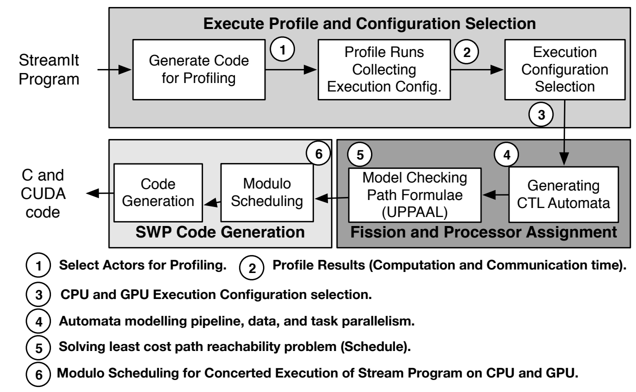

Execute Profile and Configuration Selection
StreamIt Program
Generate Code for Profiling
Profile Runs Collecting Execution Config.
Execution Configuration Selection
C and CUDA code
Code Generation
Modulo Scheduling
SWP Code Generation
Model Checking Path Formulae (UPPAAL)
Generating CTL Automata
Fissión and Processor Assignment
1 Select Actors for Profiling.
2 Profile Results (Computation and Communication time).
3 CPU and GPU Execution Configuration selection.
4 Automata modelling pipeline, data, and task parallelism.
5 Solving least cost path reachability problem (Schedule).
6 Modulo Scheduling for Concerted Execution of Stream Program on CPU and GPU.

Figure 2: Overview of compilation process, targeting stream programs on heterogenous architecture

3.4 Model Checking

Given a state machine and properties specified as temporal logic formulae, model checking aims to formally verify whether these properties are satisfied on the machine[6]. CTL formulae can be used for specifying path properties and state properties independently with path and state formula respectively.

We create the automata from the stream graph along with the cost associated with the assignment of an actor to a processing core on CPU or GPUs. The constraint (specified in CTL) is to find a reachable path with minimum cost in the model. This minimum cost reachable path is used to derive a schedule and to generate efficient code for heterogeneous architectures.

4. SCHEDULING STREAM GRAPH USING MODEL CHECKING

4.1 Overview of Compilation Process

Figure 2 shows the various phases involved in our compilation process for generating software pipelined code for heterogeneous architectures. The diagram is annotated with numbered labels to illustrate the flow. We present an overview of the steps in the compilation process below.

1. An actor is selected and annotated for profiling and its machine code is generated using the modified StreamIt compiler[17]. Modifications were carried out in-house. The annotated actor is run and its computation time on CPU and GPU cores are collected. This step is repeated for all the actors.

2. The average communication time between the CPU and the GPU is measured by running simple programs.

3. Using the results from the previous steps, the optimal number of GPU streams is computed. Stateful filters are marked to be scheduled on the CPU cores.

4. The computation automata are built for each actor. The actions for the automata are appropriately set using the computation cost and the costs for coalescing data accesses on GPUs.

5. The automata are given to a model-checker to perform the least cost reachability analysis on the paths of the model.

6. The result from the model checker is used to detrermine the stages required to build a modulo scheduled pipelined code.

Section 4.2 describes modules 2 and 3. Section 4.3 describes modules 4 and 5. Module 6 is described in section 4.5, and section 4.6 deals with code generation for CPU cores and GPU.

4.2 Stream Program and Architecture Configuration

The architecture in consideration as described earlier consists of both multi-core CPU and an NVIDIA GPU. For explanation we chose a system with two CPU cores (M1 and M2) and one GPU (G1), in which each processing unit can communicate with all others. The CPU cores act as the master from where the communication is initiated and coordinated.

4.2.1 *Optimal number of CUDA streams*

We derive the number of streams to be created on the GPUs for a streaming application based on the performance model by GóMez-Luna et. al.[9]. We use the following equations[9] where, t_K represents the kernel execution time, t_{hd} stands for the data transfer time from host to device and t_{dh} the data transfer time from device to host. Transfer times depend on the number of data items to be transmitted and on the characteristics of the bus on which the data is transferred. t_{sc} is the time required to create a stream, and is estimated for each GPU by profiling.

The optimal number of streams when the data transfer time is dominant is given by $nStreams = \sqrt{\frac{t_K}{t_{sc}}}$. The optimal number of streams, when the kernel execution time is dominant is given by $nStreams = \sqrt{\frac{t_{hd}}{t_{sc}}}$.

t_{sc} in our case varies from 0.02 to 0.06, and we use a value based on the profiled results on the GPU. This optimal number of streams serves as the number of processor that we model in our architecture graph for the GPU, instead of assuming 1000's of cores which makes scheduling infeasible.

4.2.2 *Profile Execution and Configuration Selection*

It is important to determine the resource usage and optimal execution configuration for a given stream program on both the CPU and the GPU. On the GPU this configuration is specified by the execution time of actors, number of streams, the number of threads per block and the number of thread-blocks per stream, and is achieved by the profiling the code with the help of the NVIDIA *nvcc* compiler. On the CPU, the cost of execution of actors and the cost of data transfer to and from GPU is obtained by profiling (transfer cost is obtained on a representative fixed size data and then appropriately calculated for each transfer). We use this profile data at the integrated actor fission and processor assignment stage to build our model and formulate a CTL reachability problem.

4.3 Integrated Actor fission and Processor Assignment

Let graph $M(P, C)$ represent the heterogeneous execution architecture, where $P = \{p_1, ..., p_h\}$ are the processors on the system, and $C \subseteq P \times P$ is the set of communication links between the processors.

A *parallel periodic schedule*, schedules the execution of actors on a parallel architecture and has a finite sequence of actor invocations for each processor. Synchronization between processors is necessary to begin subsequent iterations of the periodic schedule of stream graph G. The time taken by the path with the highest execution cost in the stream graph (with parallel execution of actors) is defined as the *makespan* $\pi(G, M)$.

The objective of the this Section is to find a parallel schedule with an allocation for every vertex $v_i \in V$ where $i \in \{1..|V|\}$ on some processor $p_j \in P$, where $j \in \{1..h\}$ which minimises the *makespan* $\pi(G, M)$ for the stream graph.

We use Uppaal[2] to represent our stream graph as automata, and utilise the reachability property of CTL to achieve our objective of finding the minimum makespan [14]. Uppal notations for *location* and communication between automata are used. A node marked "U" means it is an *urgent* location i.e., the transition from the location has to be taken as soon as the guard is true. Asynchronous communication among automata is represented using Milner's CCS[3] style handshake communication. This is used to represent parallel transition of automata in our example and is not to be confused with any synchronisation in the stream graph.

4.3.1 *Building Computation Models from the Stream Graph*

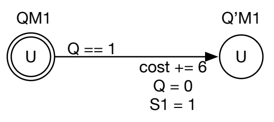
We build the automata in a way similar to that of Malik et. al.[14], but our model is simpler than theirs due to the fact that we do not include communication as a part of the model, but choose to handle it at code generation time. This reduces the number of automata generated, and thereby reduces the time for model-checking.

Figure 3(a) shows an example automaton for the filter Q scheduled on CPU processor $M2$. The label (location name in Uppaal) $QM2$ indicates the the filter and its processor assignment. The edge of the automaton describes the firing conditions and the action taken.

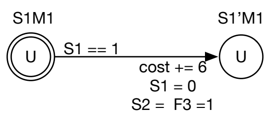
The transition Figure 3(a) is guarded by the condition QM2==1. On taking a transition, a global variable **cost** is incremented by the computation time of Q on CPU2, which in this case is 3*2, where 2 and 3 are the computation cost and the natural granularity of Q respectively. At the end, the actions also sets its guard condition to false (QM2=0), and set the next filter guard condition to true (S1M2=1) to enable further transitions. Many automata are possible for each actor because each actor may be placed on any core of the CPU or the GPU. For example, $S1$ can give rise to S1M1, S1M2, and S1G1 (see Figure 3). The source actor (here Q) is always placed onto the CPU core which is decided by the programmer. All such automata are built for each actor in V.

4.3.2 *Modelling Task Parallelism*

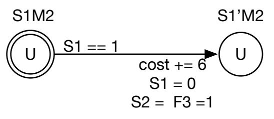
The automata in the previous sections represent sequential execution of the stream graph, where only one of the automata is free to make a transition based on its guard, and when done, it sets its own guard false and enables the



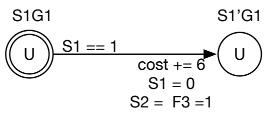
(a) actor Q on CPU core M2



(b) splitter S1 on CPU core M1



(c) splitter S1 on CPU core M2



(d) splitter S1 on GPU G1

Figure 3: Computation Automata

next automaton in the sequence in the stream graph.

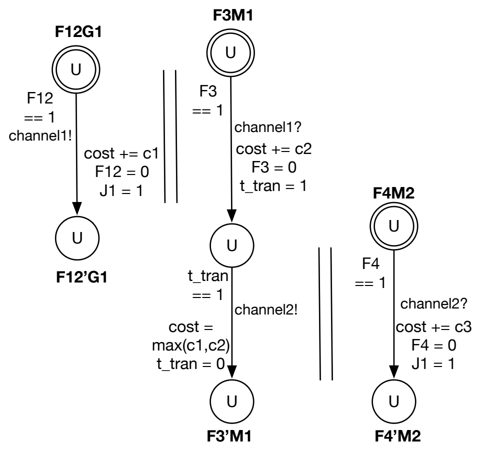


Figure 4: Modelling Task Parallelism

The three actors $F1, F2$, and $F3$ in Figure 1 can possibly run in parallel provided there are no resource constraints. $F4$ can be run only after $F3$. Let us (for simplicity) assume that $F1, F2$ being identity filter are fused together as $F12$, which is a very common optimisation for stream graphs.

Figure 4 shows an example network built in Uppaal by combining the basic automata representing the allocation of $F12$, $F3$, and $F4$, on CPU1, GPU1, and CPU2, respectively. The first automaton (F12G1) communicates with the first transition of the second automaton (F3M1) via channel *channel1*, (all communications modelled here between automata are rendezvous communications). This rendezvous communication causes the two transitions to take place together (thus allowing parallel transition of the two automata), in the process transferring the execution cost of $F12$ on GPU1 (cost1 = c1) to the other automaton. Upon completion of this rendezvous, the actions sets the guard for the second transition (*t_tran*) high. This allows the second transition of the second automaton to rendezvous with the third automaton via *channel2*. The maximum of the received values of *cost*1 and *cost*2, the execution cost, is transferred to the third automaton. F3 is a stateful actor and cannot be run in parallel with any filter in its pipeline.

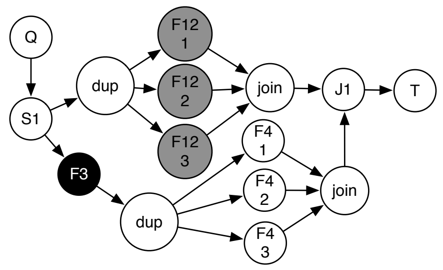
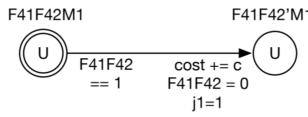
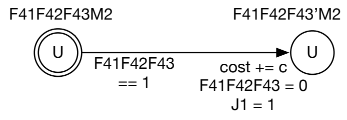


Figure 5: Modelling data Parallelism



(a) actor F41,F42 fused and mapped on CPU core M1



(b) actor F41,F42,F43 fused and mapped on CPU core M2

Figure 6: Optimal data parallelism exploitation

In other words, the two transitions of F3 (see Figure 4) could not have taken place in parallel.

4.3.3 *Modelling Data Parallelism with Integrated Fission*

Stateless filters can be replicated to utilize idle processor resources, which further increases the data parallelism available in the stream graph due to split-joins. Figure 5 shows replication of the stateless $F12$ and $F4$ filters three times, one for each processor. This technique allows utilization of all the three processors. But such simple replication scheme may lead to communication overhead. We use Algorithm 1 to replicate stateless filters judiciously by building all possible fused automata for the replicated filters so that the optimal judicious fusion of actors is obtained. Due to fusion of automata, the overall model would require less resources and time on execution, because only one of them would have been selected based on their guards). Identity actors are handled as special cases as no extra communication channel for them would be required on fusion.

Our algorithm produces lesser number of automata than that of Malik. et. al [14]. This is due to the exclusion of communication cost in our model which reduces the number of actors in the graph with replications. Figure 6 shows the generated automata for actor $F4$ which is replicated into $F41, F42$ *and* $F43$ naively and then generating automata for all their possible fused combinations. Malik. et. al [14] algorithm would have generated F41F42 and F42F43, where we clearly see that F42F43 is similar in properties to F41F42, and thus we do not generate any such extra automata.

4.3.4 *State Space and Reachability Property*

Algorithm 1 Building fused replicated filters automaton states

```
Input: Execution Architecture $M$, Modified Stream Graph $G'$
Output: A set $S$ of fused replicated filters automaton states
    for each processor    ▷ Generates minimum number of
    automata instead of all possible permutations.
 1: $S \leftarrow \phi$
 2: $i \leftarrow 0$
 3: $\mathcal{SJ} \leftarrow$ the set of all split joins in $G'$ with replicated
    filters
 4: for all $s_i \in \mathcal{SJ}$ do
 5:     $B \leftarrow$ branches in $\mathcal{SJ}$
 6:     $s \leftarrow \phi$
 7:     if $|B| \geq 2$ then                ▷ more than two branches
 8:         for all $b_j \in B$ do
 9:             for all $p_j \in P : P \in M$ do
10:                 ▷ merging states and add processor name
    as label to a new state
11:                 $s \leftarrow newstate(v \in b_j, s, p_j)$
12:                 $S \leftarrow S \cup \{s\}$
13:             end for
14:         end for
15:     end if
16: end for
```

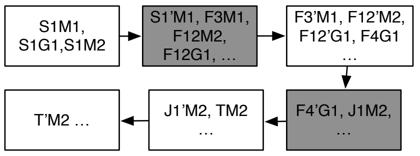

S1M1, S1G1,S1M2
S1'M1, F3M1, F12M2, F12G1, ...
F3'M1, F12'M2, F12'G1, F4G1 ...
F4'G1, J1M2, ...
J1'M2, TM2 ...
T'M2 ...

Figure 7: State transitions in UPPAAL

We use the Uppaal model checker to find the optimal makespan of the stream graph. All the constructed automata are input into Uppaal and a reachability property $E <> (FinalState$ and $cost < \infty)$ is asked to be verified. The result is that the property is satisfied. The trace generated from Uppaal gives the allocation within the states and the schedule via transitions. An example partial trace from state transitions in uppaal is shown in Figure 7. Here, splitter S1 can be possibly allocated on any of the three processing units M1, M2 or G1. However, from the transition it is clear that S1 should be allocated on M1 only, so as to enable the transition from S1M1 to S1'M1 to happen. The second state in Figure 7 highlights that F3 can be executed in parallel with F12, and that F12 can be allocated on either M2 or G1. However, the transition to the next state shows that F12 should be allocated on G1 only. Lets assume that Uppal found a trace with cost $C1$ based on the property.

We reiterate through this model-checking process with the new reachability property $E <> (FinalState\ and\ cost < C1)$, and try to find a trace whose cost is always less than $C1$. If no such trace can be found, then $C1$ is the least cost trace. Our approach converges to a solution in 8 or less number of iterations.

4.3.5 Coalescing GPU Accesses

To effectively utilise the high memory bandwidth available on GPUs, the memory accesses by the running kernel must be contiguous. Thus coalescing the access to GPU memory together with the appropriate buffer layout considerably increases the performance of the code executing on GPUs [20] [12] [18]. 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 memory bank. Formally, thread N of a warp must access an address of the form *WarpBaseAddress + N*, with *WarpBaseAddress = NewAddress modulo Number of Banks*. Such accesses by all the threads can then be coalesced into a single access.

If two actors are scheduled as producer on the CPU and as consumer on the GPU, we add the corresponding shuffling cost. If the consumer is on the CPU and producer on the GPU, we add the deshuffling cost which reverses the coalescing done for GPUs memory access. The overall cost is added on the edge of the consumer actor automaton. In brief shuffling operation coalesces the memory access when the data is to be transferred from CPU to GPU. Deshuffling reverses the Shuffle operation. Thus, our model incorporates all the necessary costs for optimal partitioning of actors onto the processing units.

4.4 Stream Graph with Feedback Loops

Stream graph with feedback loops introduce backward dependencies in the stream graph. Udupa et.al [19] do not handle feedback loops. The algorithm of Malik et.al [14] will also be unable to exploit the data parallelism with feedback loops because their algorithm for actor fusion searches for forward split-joins whereas feedback loops in StreamIt have join before a split. Our model handles feedback loops in the stream graph easily. Figure 8 shows an example stream graph with a feedback loop.

Figure 8(a) and 8(b) show a StreamIt program for fibonacci series generation using feedback loop, and its corresponding stream graph respectively. Though the code is inefficient, it demonstrates the handling of a feedback loop in stream graphs with our approach.

We demonstrate the assignment obtained for the stream graph on two processors M1 and M2. Figure 8(c) shows the computation automata for the actors in the stream graph. Figure 8(d) shows how our approach can easily generate task parallel automata even when the join precedes the split in the stream graph. In this example, we assume allocation on two processors. For more than two processors, our Algorithm 1 obtains fused copies for the actors.

The trace as obtained from the model checker is shown in Figure 8(e), and the corresponding allocation of actors of the stream graph onto the processors is shown in Figure 8(f).

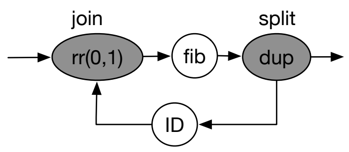
Here we have shown all the automata for the example and it is easy to see how the model checker has the option to choose from the all the possible automata configurations and obtain a least cost reachable path in the model. Later stages in code generation are described in the following sections.

4.5 Stage Assignment

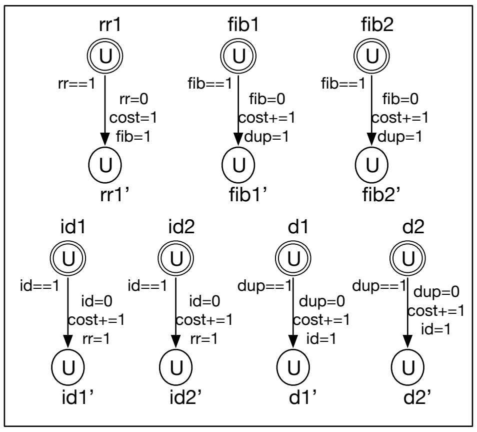
The processor assignment obtained in the previous section does not specify how they overlap in time. To honour data dependencies the execution of the actors corresponding to a single iteration of the stream graph is grouped in *stages. This technique helps us avoid communication costs in modelling the partitioning problem* We use predicated staging to

```
void -> int feedbackloop fibonacci {
    join roundrobin (0, 1);
    body int -> int filter {
        work pop 1 push 1 peek 2 {
            push (peek (0) + peek (1));
            pop();
        }
    };
    loop Identity<int>;
    split duplicate;
    enqueue (0);
    enqueue (1);
}
```

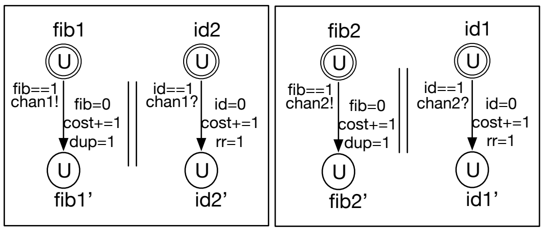
(a) StreamIt code for fibonacci series with feedback loops



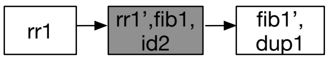
(b) Stream graph for feedback loop fibonacci code



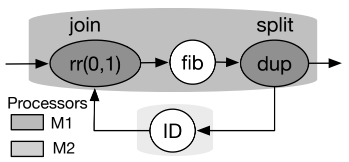
(c) Computation automata for feedback loop example



(d) Task parallel automata

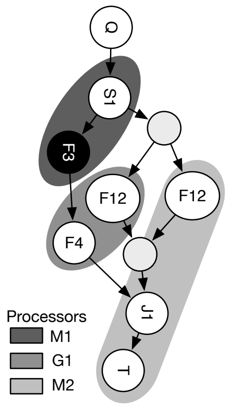


(e) Trace obtained from model checker

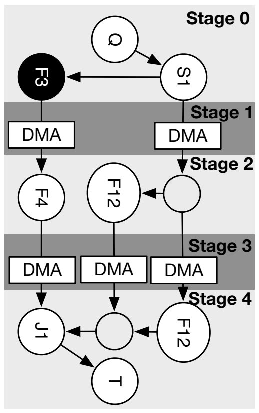


(f) Corresponding processor assignment

Figure 8: Feedback loop example



(a) Processor assignment as obtained from the model checker



(b) Stage assignment

Figure 9: Processor and Stage Assignment of Actors

assign filters to streams so as to setup a pipeline[15] similar to modulo scheduling. We do not need to traverse the graph in any topological ordering as needed by the previous approaches[19][12], since we already have a trace of possible state transition from the model checker.

Based on the trace and its processor assignments, the following rule (from [19]) is used to assign stage numbers to actors and to add code for the appropriate shuffle and reshuffle operations of data when actors are placed across processors. For the actors a_i and a_j where a_i is the producer and a_j the consumer, the stage number is assigned by one of the following rules:

1. if a_i is assigned to the CPU and a_j is assigned to the GPU or vice versa, then stage(a_j) >= stage(a_i)+2.
2. If both a_i and a_j are assigned to the CPU or to the GPU, then stage(a_j) >= stage(a_i) + 1.
3. if the edge connecting a_i and a_j requires a shuffle or deshuffle operation, the the stage of a_j as obtained from the above rules is further incremented by one.

Rule 1 enables insertion of a DMA operation in the intermediate stage and makes sure that actors across processors have stage number separated by at least two. Rule 2 ensures that the producer-consumer dependency between the actors is reflected by their stage numbering. And, Rule 3 enables inserting an extra intermediate operation along with DMA transfer if required to shuffle or deshuffle data. Figures 9(a) and 9(b) show an example of stage assignment and DMA code insertion for a single iteration of the example stream graph.

It should be noted that our stage assignment algorithm does not fail even if there is a loop in the stream graph, as we do not rely on any topological sorting on the stream graph to determine the producer-consumer dependence among actors.

4.6 Code Generation

Based on processor allocation, our modified StreamIt compiler generates C as well as CUDA code for the parallel version of the stream program. We schedule streams on the

```
void work() {
 int stage[N] = {0};
 stage[0] = 1;
 for (i=0; i<max_iter+N-1; i++) {
  if (stage[N-1]) {

    for (int i = 0; i < number_of_streams; ++i)
      cudaMemcpyAsync(inputDevPtr + i * size,
         hostPtr + i * size,
          size, cudaMemcpyHostToDevice, stream[i]);
    for (int i = 0; i < number_of_streams; ++i)
       actor1<<<num_blocks / number_of_streams,
          num_threads, 0,
          stream[i]>>>(outputDevPtr + i * size,
          inputDevPtr + i * size, size);

 }if (stage[N-2]) {
 }
 ...
 if (stage[0]) {
 }
 //wait_for_dma_completion
 for (int i = 0; i < number_of_streams; ++i)
     cudaMemcpyAsync(hostPtr + i * size,
        outputDevPtr + i * size,
        size, cudaMemcpyDeviceToHost, stream[i]);
  // start epilogue
 if (i == max_iter-1) stage[0] = 0;
 // Shift-left staging predicate
 for(j=N-1; j>=1; j-- )
      stage[j]=stage[j-1];

 cudaThreadSynchronize();
```

Figure 10: Modulo scheduled CUDA stream calls from the Host CPU Code

GPU (CUDA streams) which is called from the main C program on the host (CPU). A sample modulo schedule with CUDA stream calls in the code for a single actor in a stream is shown in Figure 4.6. The *number_of_streams* is obtained as explained in Section 4.2.1. Here the assumption is that the actor code has been scheduled on all the streams on the GPU, and that the data structures and variables in the code have been appropriately managed so that a stable stream could be scheduled onto the GPU, executing concurrently with the CPU code. The array stage functions similar to the *staging predicate*[15] of the modulo scheduling, and its size (N) is the maximum number of stages. The main loop starts with only the first stage active. The *if* conditions that test different elements of stage ensure only actors assigned to a particular stage are executed. The last part of the loop shifts the elements of the array stage to the left, which has the effect of filling up the software pipeline. When all the iterations are done, the pipeline is drained by shifting a 0 in the last element of the stage array.

The code in the active stages are the kernel calls for the corresponding work function of the actor in the stream graph. The DMA operations are shown by the *cudaMemcpyAsync* function in the code, and these calls as the names suggest, are non-blocking. Thus all the DMA operations are put in queue before any computation is started, which provides for

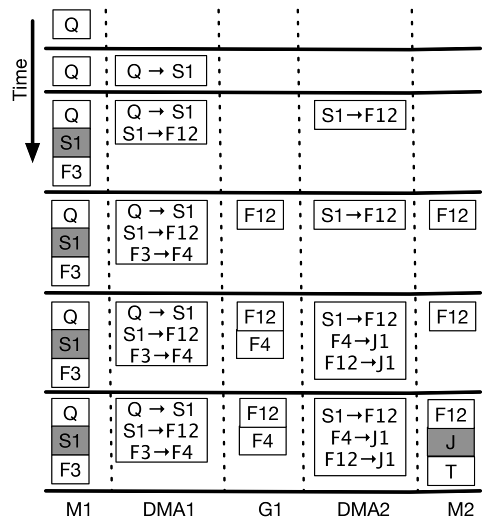


Figure 11: Modulo scheduling running on 3 processors

a maximal overlap of DMA with the work functions of the actor. Barrier synchronization is executed to ensure that the current iteration is finished on all the streams on the GPU, before the next iteration begins.

Conceptually Figure 11 shows the timeline of execution of code on three processors, and any of them can be either CPU or GPU. The steady state execution (as shown) starts from the fifth iteration in Figure 11, where all the computation and the DMA transfers are simultaneously active. All the previous four phases form the prologue of the modulo scheduled software pipeline. All the corresponding DMA transfers are activated before any actor starts execution on the processors. *This overlap of computation and communications allows us to avoid communication costs in our model built in Section 4.*

5. EXPERIMENTAL EVALUATION

We have implemented our approach as an extension to the StreamIt compiler version 2.1.1[17]. Our compiler generates both C with pthreads and CUDA code for multicores and GPUs respectively. The code on GPUs are compiled with NVIDIA *nvcc* compiler with the CUDA 5.5 toolchain. The results reported in this section were performed on a machine with 2.4 Ghz 12 core Xeon E5646 CPU with 16 GB Ram and a NVIDIA TESLA 2075 GPU, where each GPU has 448 CUDA cores (14 Multiprocessors $\times$ 32 CUDA Cores/MP).

The StreamIt benchmarks used for our experiments is described in Table 1. Most of these benchmarks are from signal processing domain. Most of the StreamIt benchmarks do not scale well beyond 8 cores. [16] provides a comprehensive details of the benchmarks and their characteristics. We implemented the heuristics-based syngergistic software pipelining of Udupa et.al [19] and the approach of Malik et.al.[14] with all the communication costs incorporated into the model, on our target (heterogeneous architecture) for comparison with our method described in Section 4. The algorithm of [19] did not work for a system with only CPU or GPU in it. We modified the algorithm to obtain a partitioning for just multi-cores to compare with our results. Malik et.al.

Table 1: Characteristics of the Benchmarks

Benchmarks	Actors			Description
	Total	Stateful	Peeking	
Bit	82	0	0	Batcher's Bitonic Sort
BitR	452	2	0	Bitonic Sort recursive implementation
CV	54	2	34	Vocoder Implementation
DCT	22	18	16	Discrete Cosine Transform
DES	375	180	1	DES Cryptography algorithm
FFT-C	26	14	0	Fast Fourier Transform Coarse grained
FFT-F	99	0	0	Fast Fourier Transform Fine grained
FB	53	34	16	Filter Bank for multirate signal processing
FM-R	67	23	22	Software FM Radio with Equalizer
MM	52	2	0	Block Matrix Multiplication
MPEG	39	7	0	Subset of MPEG2 Decoder (backend)
TDE	55	27	2	Time Delay Equalization phase from Ground moving Target indicator

[14] does not produce any code for any architecture as they are only interested in obtaining an optimal makespan. We extended the work of [14] as well, with a code generator to compare against our results.

Stateful filters are always placed on the CPU cores, as they introduce dependency between iterations. CPUs are well suited to handle such dependent iterations.

We abbreviate our method as MC-SWP to denote model checking based software pipelining of stream programs. Other implementations are represented by their authors' names in the results. All our comparisons of speedup are against a single-threaded CPU code generated by the cluster backend of the StreamIt compiler compiled with *gcc*.

5.1 Comparing makespan

Table 2 shows the makespan values for benchmarks used in our experiments. The architectural configuration to obtain makespan was was assuming 4 CPU cores and 8 GPU Streams. We do not take individual GPUs into consideration as explained in Section 4.2.1, and rather take streams as the computational resource in building our CTL-based model. The heuristic partitioning approach of Udupa et.al [19] is not a model-checking based approach, and it gives us an II(Initiation Interval) for software pipelining. We use the makespan as our II in our modulo scheduled software pipelined code generator.

The makespan obtained by Malik's approach is optimal, as it takes both the computation and communication costs into account, whereas our approach does not include communication costs into the model. Our approach has a lower makespan than that of the optimal makespan, whereas, Udupa's heuristics give a larger makespan, indicating degradation in the execution time even for one iteration. A makespan lower than the optimal one does not imply incorrect code, because a barrier is executed at the end of every iteration of the software pipeline.

Table 2: Makespan

Benchmarks	Makespan (ns)		
	MC-SWP	Malik et.al.	Udupa et.al.
Biti	72570	77202	84292
BitR	105262	116958	147102
CV	8587960	8853568	10373877
DCT	1524609	1621925	1787428
DES	371921	413246	464369
FFT-C	317839	327669	413723
FFT-F	394579	419765	454031
FB	636420	707133	801904
FM-R	199727	205905	222543
MM	1197292	1273715	1455675
MPEG	1675072	1861191	2033879
TDE	14065412	14500425	16111583

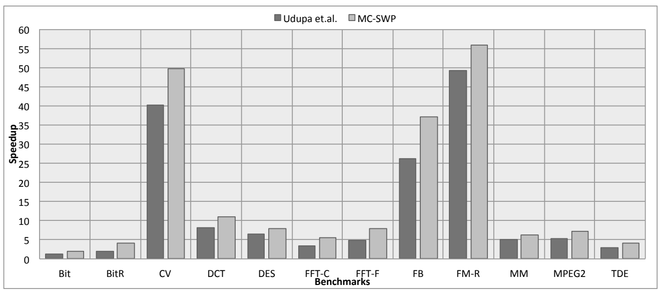


Figure 13: Speedup on CPU + GPU architecture normalised to single StreamIt CPU.

5.2 Comparison of MC-SWP with Optimal and Heuristics-based Partitioning

Figure 12 shows the speedup obtained with the benchmarks using Udupa's heuristics, Malik's approach, and our scheme. Our scheme results in a maximum speedup of 8.25X and a geometric mean speedup of 4.67X over all the benchmarks for 8 cores, wheres Udupa's scheme achieves a maximum speedup of 7.77X and a geometric mean speedup of 2.93X over all the benchmarks for 8 cores.

Malik's scheme with our code generation phase has upto 8.17X speedup and a geometric mean speedup of 4.68X over all benchmarks for 8 cores. This shows that an efficient code generation strategy can offset some of the constraints ignored while building a model for obtaining an optimal makespan, and thus a simpler model can provide the same speedup at runtime.

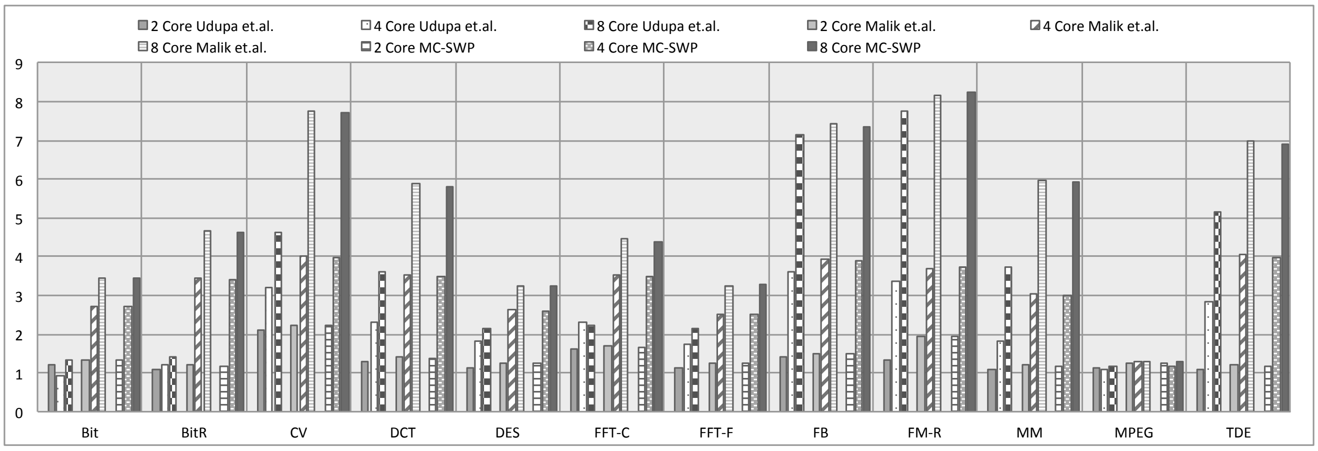


Figure 12: Speedup on 2, 4, and 8 cores CPU normalised to single StreamIt CPU.

Figure 13 shows the speedup comparison on systems with both CPU and GPU together. Since Malik's approach produces a result similar to ours, we do not show the results with Malik's approach. Here again our scheme is better than Udupa's, resulting in a maximum speedup of 55.86X and a geometric mean speedup of 9.62X over all the benchmarks for 4 CPU cores and *8 GPU streams* (see the sections 5.3 and 5.4). Udupa's scheme provides upto 49.32X speedup and a geometric mean speedup of 6.76X over all the benchmarks for 8 cores.

5.3 Comparison of MC-SWP with Stream Graph on GPUs without Streams

Figure 14 shows the speedup for five benchmarks when the stream on GPUs are not used to schedule the code. Not using streams places several restrictions on code-generation. Without streams, only one kernel can be in flight on the GPU, thus underutilzing the GPU resources as there are not enough computations in one actor to use the GPU resources to its optimum level.

To make a fair comparison we enabled coalescing and de-coalescing for data transfer to and from GPUs respectively. The remaining schedulable actors were run on the CPU following SAS(Single Appearance Schedule)[11]. To have enough work for each actor while running on GPUs we made the whole stream graph run for 10,000 iterations, and thus each kernel was operating on $10,000 \times \eta_k$ data elements for actor k in the stream graph.

This comparison clearly makes the case for using streams on GPUs in order to enable *concurrent kernel execution* on GPUs and to enable effective coarse grained software pipelining of stream graphs on heterogeneous architectures.

5.4 Discussion

Some of the benchmarks such as Bit and BitR perform poorly with a speedup of 2.10 and 4.10 (resp.) on the CPU-GPU combined execution architecture because these applications are extremely bandwidth-sensitive. We see no gain with Malik's optimal strategy as well.

Our model does not have constraints regarding generation of code for any specific architecture, as the model is portable and usable across architectures. Incorporating the appropriate costs into the model could take care of most of the architectures.

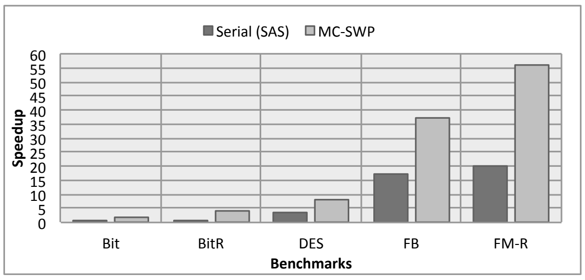


Figure 14: Speedup comparison without streams(serial) and MC-SWP.

Streams on GPUs are as important as other computation factors to achieve the best performance. The code generation technique of Udupa[19] does not use streams and hence only one kernel executes on the GPU at a time. We have used our code generator with their heuristics in the comparisons in section 5.2. Otherwise, their speedup would have been much lower. Our work is the first one to use streams optimally on GPUs to exploit all the parallelism exposed by the stream graphs.

It is to be noted that Malik et. al has no code generation stage and their comparison is only on the quality of the makespan. But, we show that using optimal makespans without good code generation strategies may not always result in optimal performance.

6. CONCLUSIONS

Stream programming models naturally expose the parallelism in applications to the programmers, and a system to map these onto heterogeneous architectures have still not been fully explored. In this paper we present a model-checking based framework for statically scheduling stream programs on heterogenous architecture having both CPU and GPUs. We produce a schedule which provides an efficient mapping onto these architectures and fully utilises the available resources. We use CUDA streams on NVIDIA

GPUs, where the optimal number of streams is decided using a profile-based approach.

To best of our knowledge our approach is the first one which utilises model-checking in a compiler that schedules and generates code for heterogeneous architectures. Our approach provides a speedup of upto 55.86X and a geometric mean speedup of 9.62X over a single threaded CPU on StreamIt benchmarks.

7. REFERENCES

[1] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA '10*, New York, NY, USA, 2010.

[2] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[3] S. Brookes. Deconstructing ccs and csp asynchronous communication, fairness, and full abstraction, April 2000.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*, 2004.

[5] P. M. Carpenter, A. Ramirez, and E. Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 57–66, New York, NY, USA, 2009. ACM.

[6] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[7] CUDA. Cuda toolkit documentation, http://docs.nvidia.com/cuda/index.html.

[8] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Orchestration by approximation: Mapping stream programs onto multicore architectures. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 357–368, New York, NY, USA, 2011. ACM.

[9] J. GóMez-Luna, J. M. GonzáLez-Linares, J. I. Benavides, and N. Guil. Performance models for asynchronous data transfers on consumer graphics processing units. *J. Parallel Distrib. Comput.*, 72(9):1117–1126, Sept. 2012.

[10] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, pipeline parallelism in stream programs. In *ASPLOS*, 2006.

[11] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *Conference on Languages, Compilers, Tools for Embedded Systems (LCTES)*, 2003.

[12] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Conference on Programming Language Design and Implementation (PLDI)*, 2008.

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

[14] A. Malik and D. Gregg. Orchestrating stream graphs using model checking. *ACM Transactions on Architecture and Code Optimization (TACO*, 10(3), Sept. 2013.

[15] B. R. Rau. Iterative Modulo Scheduling. Technical Report HPL-94-115, HP Labs, November 1995.

[16] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 365–376, New York, NY, USA, 2010. ACM.

[17] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, France, 2002.

[18] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *CGO*, Washington, DC, USA, 2009.

[19] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Synergistic execution of stream programs on multicores with accelerators. In *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 99–108, New York, NY, USA, 2009. ACM.

[20] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.