Automated architecture-aware mapping of streaming applications onto GPUs

Andrei Hagiescu* Huynh Phung Huynh† Weng-Fai Wong* Rick Siow Mong Goh†
*School of Computing, National University of Singapore
†A*STAR Institute of High Performance Computing, Singapore
{hagiescu,wongwf}@comp.nus.edu.sg, {huynhph,gohsm}@ihpc.a-star.edu.sg

***Abstract*—Graphic Processing Units (GPUs) are made up of many streaming multiprocessors, each consisting of processing cores that interleave the execution of a large number of threads. Groups of threads - called *warps* and *wavefronts*, respectively, in nVidia and AMD literature - are selected by the hardware scheduler and executed in lockstep on the available cores. If threads in such a group access the slow off-chip global memory, the entire group has to be stalled, and another group is scheduled instead. The utilization of a given multiprocessor will remain high if there is a sufficient number of alternative thread groups to select from. Many parallel general purpose applications have been efficiently mapped to GPUs. Unfortunately, many stream processing applications exhibit unfavorable data movement patterns and low computation-to-communication ratio that may lead to poor performance. In this paper, we describe an automated compilation flow that maps most stream processing applications onto GPUs by taking into consideration two important architectural features of nVidia GPUs, namely interleaved execution as well as the small amount of shared memory available in each streaming multiprocessors. In particular, we show that using a small number of compute threads such that the memory footprint is reduced, we can achieve high utilization of the GPU cores. Our scheme goes against the conventional wisdom of GPU programming which is to use a large number of homogeneous threads. Instead, it uses a mix of *compute* and *memory access* threads, together with a carefully crafted schedule that exploits parallelism in the streaming application, while maximizing the effectiveness of the unique memory hierarchy. We have implemented our scheme in the compiler of the StreamIt programming language, and our results show a significant speedup compared to the state-of-the-art solutions.**

***Keywords*-GPU, stream processing, StreamIt**

I. INTRODUCTION

Stream-based processing is an important domain of applications. Streaming programming languages ease the expression of parallelism in such applications [1]. Fine-grained computation is encapsulated in small code units, called *filters*, with small data sets. However, this model requires adjacent filters to communicate through memory. At the start of a filter, data is read from memory, and at the end of a filter's execution, the results are written to memory, all in a phased manner.

As hardware platforms, GPUs have gained good traction in general purpose computing, and in particular, high performance computing [2][3]. Various programming frameworks and run-time environments have been proposed for this purpose [4][5]. Since its early use in general computing, streaming programming languages have also been used in programming GPUs [6]. A GPU is made up of a number of streaming multiprocessors (SM), which in turn consist of a number of execution cores running in SIMD mode. Blocks of parallel software threads run on each of the available SMs. Typically, there are much more software threads than there are cores. In order to schedule the many threads on SMs, they are statically grouped into scheduling units – called 'warps' and 'wavefront', respectively, in nVidia and AMD literature[1]. Warps execute in lockstep, and if one or more threads in a warp block, the entire warp has to block. A hardware scheduler will then select another ready warp for execution.

GPU programmers are generally encouraged to expose as much parallelism as possible so that the hardware scheduler can utilize more ready threads to hide potential stalls [5]. However, there is a cost to having too many threads – increasing the number of threads diminishes the number of registers allocated to each thread, potentially causing spills to global memory. Besides this trade-off, a second hidden penalty is also often overlooked. More threads reading their input, output and local data stored in the global memory lead to more memory traffic, potentially exceeding the available memory bandwidth. Jittery, application-specific memory access patterns (such as intensive memory accesses at the beginning of a filter in order to read the inputs) can further exacerbate these problems.

Each SM in a GPU contains a small but very fast on-chip memory that is shared among all the threads in the SM. We shall refer to this as '*SM memory*'[2]. Due to its size (i.e. 16KB for each SM in the nVidia Tesla 10-series and 48KB in the 20-series 'Fermi'), and because the large number of independent threads that will run in a SM leads to large memory footprints, it is typically not well utilized. This paper describes an automated compilation flow that maps and orchestrates StreamIt programs for execution on GPU processors, avoiding the issues mentioned above, as well as maximizing the effectiveness of the SM memory. At the heart of the flow is a mapping scheme that is based on a static GPU performance model derived from the GPU specifications. Our key idea is that we can take advantage of the structure of streaming applications and move slow *global* memory accesses from compute threads into another class of threads

[1]As our proposed scheme is more suitable for nVidia-class of architectures, we shall refer to such a group of threads as a 'warp' in this paper.

[2]We find the nVidia way of refering to this as 'shared memory' potentially confusing.

so that the former can run at full speed while the latter's number is kept to a level that is just sufficient to meet the former's demands. Our flow generates two kinds of threads from a StreamIt input program: specialized *memory access* ($\mathcal{M}$) threads and *compute* ($\mathcal{C}$) threads. $\mathcal{M}$ threads transfer data sets from global memory to the fast SM memory. $\mathcal{C}$ threads compute instances of the stream graph to obtain results locally inside each SM by using the data sets loaded earlier by $\mathcal{M}$ threads. We also propose to constrain the number of $\mathcal{C}$ threads such that they work exclusively with the SM memory. These are major departures from the norm of using large number of homogeneous parallel threads in programming GPUs. Our results show that these counter intuitive measures can yield significant speedups compared to more traditional approaches of mapping streaming applications to GPUs.

Section II places this work in the context of other mapping efforts from StreamIt to parallel platforms and, in particular, a previous effort to map StreamIt to GPU. Section III provides an overview of the current GPU architectures, and gives the intuition regarding how StreamIt programs can be efficiently mapped. Section IV describes the mechanisms that we implement to provide an automated translation and orchestration of the stream programs onto GPU. The performance of this scheme depends on deriving a small memory footprint (Section V). Using our scheme, we characterize in Section VI the performance achieved onto several GPUs from nVidia. We then propose a performance model that can drive our automated mapping flow. Finally, Section VII presents our results achieved on various platforms and shows the speedup we obtained with respect to previous implementations.

II. Related Work

The parallelism exposed by the streaming language StreamIt makes it a natural candidate for programming multicores [7], or parallel architectures such as Cell [8] and Raw [1]. Streaming languages have also been previously mapped to GPU platforms [9][6]. It is built on top of the synchronous dataflow model [10], with filters repeating in a static schedule. The stream graph is usually partitioned into kernels and distributed between the processing cores (or SMs in the case of GPUs) with filters in the graph communicating via memory. However, on GPUs where fast caches or dedicated communication are often missing, the overhead of accessing global memory often limits performance. To reduce run-time overhead, communication between SMs executing different kernels has to be deferred until a large amount of data is processed locally. As a result, the latency of executing the stream graph is not improved (even though throughput may be improved) despite the use of pipelining.

We take an entirely different approach in mapping streaming graphs to GPUs. Instead of partitioning, we execute multiple instances of the entire stream graph in parallel on each SM, taking care to adjust the number of parallel threads to match the resource constraints. The aim is to achieve a balance between the number of GPU threads, the layout of the SM memory, and memory bandwidth consumption that will maximize performance. Selecting the right number of parallel threads and the location of frequently used data is not trivial [11]. One well-known approach that boosts performance is to prefetch data from global memory to SM memory [12]. This is also the approach taken by other high-level language translations to CUDA and OpenCL [13][14][15]. However, to the best of our knowledge, our work is the first to apply a variant of double buffering to fully parallelize prefetching with computation.

Because the amount of SM memory is limited, we are also interested in reducing the footprint of the working set of each stream execution. There are two complementary techniques. One relies on caching transformations for StreamIt that have included narrowing the memory requirement through modulation or copy-shift [16]. In addition, temporary buffers can be overlapped during the computation. Optimal algorithms have also been proposed for compiler management of scratchpad memory [17]. Our approach is based on the copy-shift method, adapted to the way our stream graph executions share a common memory. Furthermore, it is a heuristics that is near-optimal but completes in linear time.

III. Background and Rationale

A. StreamIt language

In StreamIt, programs are described hierarchically at a conceptual level, where the leaf node is a *filter*, and filters can be combined into *pipelines*. The flow can be distributed in parallel paths using *splitters* and *joiners*. Filters are essentially C code with special constructs to access input and output. Filter communication is done through special input and output channels. Producer (consumer) access to the output (input) channel is realized through *push* (*pop*) constructs. All the input and output rates are statically defined. The compiler can therefore determine a static sequential schedule through which it can iterate to consume all the input data. Multiple copies of the entire schedule can be executed in parallel if filters do not maintain internal state. An additional feature in StreamIt is the mere inspection of a channel through *peek* constructs. A filter can peek into more data than it consumes during the current firing. This allows for structured data dependencies between multiple filter firings, and helps avoid the need for stateful filters in many situations. Our parallel mapping schemes support peeking filters, but not stateful ones.

B. Architecture-aware mapping onto GPUs

GPUs are massively parallel processors. The current architectural trend points to an increase in the number of threads supported in each SM so as to match the execution rate of the increasing number of processing cores. Current GPUs divide the thread pool of each SM into warps. For the current generation of nVidia GPUs, a warp consists of 32 threads. The threads belonging to a warp execute in parallel but in lockstep, and any intra-warp control flow discrepancies will lead to serialized executions. A hardware scheduler selects a warp for execution and dispatches the threads to the execution cores. At each instruction issue interval, the scheduler can select a different warp and dispatch it to the same execution cores even before the previous warp finishes processing. Thus, while there are a large number of parallel threads, the threads are actually interleaved onto a limited number of execution cores at the granularity of a warp.

The key to good performance is to always have warps that are ready for execution when the hardware scheduler attempts to select one. Two factors can stall the execution of a warp: the first is the latency of the execution cores (typically 22 cycles), and the other is the latency of the global memory access (around 400 cycles). For example, to hide the latency of the execution cores, nVidia suggests 6 such warps on older devices of capability 1.x and 11 on devices of capability 2.x [5]. As the global memory access is an order of magnitude slower, the number of threads required to completely hide this latency will exceed the maximum number of threads that can be supported by the hardware if all the threads concurrently require access to global memory. Unfortunately, this pattern is exhibited by many stream processing applications through filters, their basic processing units. Typically, filter execution is phased: (1) reading the data set from memory, (2) performing the computation, and (3) writing it back to memory to pass it onwards to the next processing filter. Moreover, the ratio of computation to communication is usually small. Therefore, if the filter's input and output are stored in global memory, filter instances will spend most of the time on memory accesses. It is therefore advantageous to bring data onto the SM memory shared by the threads. This is so that the filters can process the prefetched data set at a much faster rate. Unfortunately, the memory latency still cannot be completely hidden, and having threads do data prefetching before computation will still result in the computation section of the code waiting on the prefetching section most of the time.

In our proposed approach, we introduce two classes of threads: memory access ($\mathcal{M}$) threads and compute ($\mathcal{C}$) threads. $\mathcal{M}$ threads perform prefetching for the next stream execution while $\mathcal{C}$ threads execute on data fetched by the $\mathcal{M}$ threads into the SM memory during the previous stream execution. Intuitively, because the $\mathcal{C}$ threads will always access SM memory, they will always be ready for execution, while the $\mathcal{M}$ threads will be scheduled from time to time to initiate more parallel memory transfers.

Due to the architectural constraint that only threads in the same SM can communicate through the fast SM memory, our entire stream processing flow must reside in the same SM. It is replicated on all the other SMs to fully utilize the GPU. Since StreamIt exposes the potential massive parallelism within applications, we also map the parallelism available in each stream execution to multiple threads inside the SM.

IV. Mapping StreamIt to GPU

A. Mapping flow

Our automated mapping flow applies a sequence of code transformations in order to: (1) match the large number of parallel threads that can be handled by the hardware, (2) cluster the memory transfer operations with large latency into dedicated threads, (3) transform the data flow based on the fine-grained parallelism exposed by StreamIt, and (4) apply a novel buffer manipulation scheme that replaces the one used by StreamIt compiler for inter-filter communication.

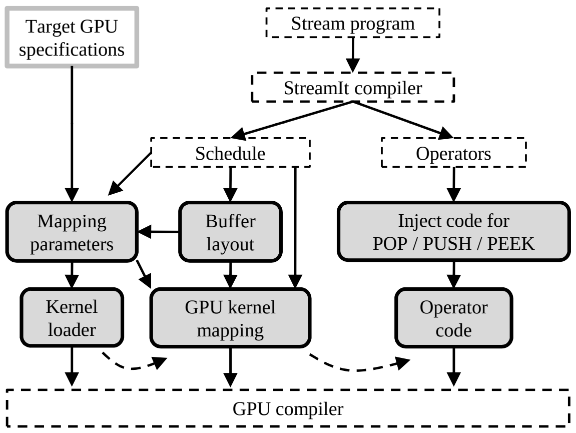
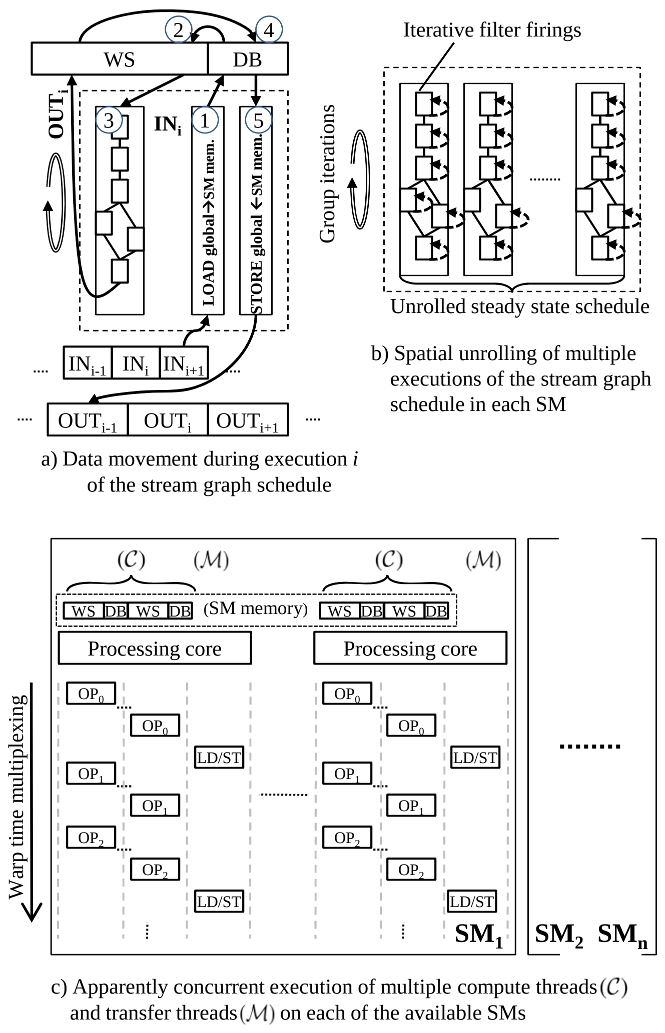


Fig. 1. Our automatic mapping flow.

We implemented our mapping flow, shown as grey boxes in Figure 1, as an extension to the back-end of the StreamIt compiler. It generates C code that can be compiled by the standard GPU compiler. The StreamIt compiler flattens the hierarchical stream program to a set of base *operators* (filters, splitters and joiners). It also produces a schedule that consists of a sequence $\mathbf{E}$ of operators, and the number of times they are executed (*fired*). Note that multiple firings may be necessary, because filters are allowed to have non-matching input and output rates and hence the elements produced by one filter's firing may require multiple firings of the consumer filter. Apart from the initialization portion, the resulting schedule consists of a steady state component that can be executed as many times as required to completely process the given input. At this point, the schedule generated by StreamIt is sequential, targeting single threaded execution. From this point on, our mapping extension takes over.

We analyze the requirements of each operator in the schedule and produce a compact buffer layout (detailed in Section V) that can eventually be realized in the fast SM memory. Once this buffer size is known, we can statically determine additional mapping parameters such as the number of stream schedules that are to execute in parallel, the number of $\mathcal{C}$ threads supporting the execution of each stream schedule, and the number of dedicated $\mathcal{M}$ threads accessing global memory. To determine the mapping parameters, access to the stream schedule structure and to the specification of the target GPU is also required. Finally, we use the derived mapping parameters to build two components: (1) a kernel loader which will run on the CPU and will coordinate the memory allocation and configuration, (2) the GPU kernel code that executes the mapping described in Section IV-B. In addition, the push, pop and peek primitives of each operator are replaced by code that perform the correct accesses of the working set buffer in SM memory. The C code implementing the operators, together



a) Data movement during execution i of the stream graph schedule

b) Spatial unrolling of multiple executions of the stream graph schedule in each SM

c) Apparently concurrent execution of multiple compute threads ($\mathcal{C}$) and transfer threads ($\mathcal{M}$) on each of the available SMs

Fig. 2. Parallel memory access and orchestration of the stream graph.

with the kernel and its loader are given to the GPU compiler to obtain the final executable.

B. Mapping to parallel threads

The CPU host allocates input and output buffers for the stream graph in the off-chip global memory of the GPU. Current GPUs are capable of concurrent code execution and host memory transfer, and hence we assume the data transfer from the host CPU to the GPU incurs no penalty. However, because global memory accesses impact the efficiency of the computation inside the GPU, we chose to avoid any unnecessary partitioning of the stream graph into multiple components residing on different SMs, as this would imply having to connect these partitions through global memory buffers. Instead, our approach is to keep the number of partitions to the minimum, and execute an instance of the entire steady state schedule of the stream graph on each SM.

Figure 2a shows the execution model of the stream graph in the GPU. Let i be the current execution of the steady state schedule. In SM memory, we allocate a working set buffer (WS) that will hold the inputs, outputs, as well as the buffers between operators for one execution of the schedule. In addition, we require a second, smaller, buffer, DB, in SM memory. It is an intermediary buffer that is large enough to hold all the stream graph inputs, or outputs, whichever is larger. Our *double buffering* prefetch scheme works as follows. Let IN_i and OUT_i denote the input and output of execution i, respectively. During execution $i-1$, IN_i is brought into the buffer DB (step ①). Before the start of execution i, IN_i is copied from DB into the input space of WS (step ②). After the completion of this copying, execution i may begin (step ③). OUT_i would reside in WS at the end of execution i. Concurrent with execution i in step ③, IN_{i+1} is brought into the buffer DB for the next execution. At the end of execution i, IN_{i+1} is copied from DB into WS replacing IN_i, after which, OUT_i is copied from WS into DB (step ④). Execution $i+1$ then begins. Concurrent with execution $i+1$, OUT_i is written back to global memory (step ⑤). This last step is interleaved with the prefetching of IN_{i+1} so that DB can be reused.

The above describes what happens in one instance of the steady state schedule. We further unroll and execute in parallel a group of W instances of the steady state schedule. Each of these executions store its local data into a separate working set buffer allocated in the fast SM memory. These executions are mostly independent except for peeking (which will be discussed later), and suitable for a parallel orchestration as described in Figure 2b. Each steady state schedule includes a sequence of filter *firings* that may be iterative. We shall call one complete processing of a stream graph an *execution* of the steady state schedule. We map each schedule execution to one or more $\mathcal{C}$ threads. We also iterate over the group of W parallel executions as many times as necessary to process all the application's inputs. We shall call a pass over the group of W executions a *group iteration*. Each SM is assigned a different part of the input and output stream in sequence. In particular, for SM_1, this sequence number starts from the beginning of the stream in global memory. All the SMs will compute the results for distinct portions of the input stream, and the access offsets in these streams are known and computed by the loader before the kernel launches.

Furthermore, the double buffering mechanism described in Figure 2a can be refined for a group of parallel executions. Loading as well as storing to global memory are performed by a set of parallel $\mathcal{M}$ threads that combine the load and store operations corresponding to all executions. Let F be the number of $\mathcal{M}$ threads. We ensure that $\mathcal{C}$ threads and $\mathcal{M}$ threads are allocated to distinct warps. They therefore execute in an interleaved manner as shown in Figure 2c. In general, $\mathcal{C}$ threads will always be available for execution, as their data dependencies are satisfied from registers or SM memory. $\mathcal{M}$ threads, however, issue long latency global memory operations, and are scheduled only sporadically. The intuition is that by adjusting the number of $\mathcal{M}$ threads (F), and $\mathcal{C}$ threads (W), we can completely hide the latency of the global memory accesses.

The *steady state schedule*, **E**, is an ordered sequence of stream operator firings that consumes a set of inputs, and eventually generates a set of results. The amount of intermediate data obtained during these executions may require more memory than the IN and OUT buffers. Additional buffers are also found in WS that are used for operators in the graph to communicate with one another. Let the total memory requirement for the WS be L_W. The size of the secondary buffer DB, on the other hand, is $L_D = \max(\text{size(IN)}, \text{size(OUT)})$.

For non-GPU multicore mappings, each core executes a single instance of the schedule, and has a large amount of memory available. The StreamIt compiler offers a feature that may fuse filters only to tune their working set and buffers to the cache size. Nevertheless, as the buffers are not reused, there was no effort to optimize the memory resource usage over the entire graph. For our work, the buffer requirement is critical as it dictates how many parallel executions of the graph we are able to run because we need to be able to store the complete working set in SM memory. We describe our algorithm to determine a compressed WS buffer layout in Section V.

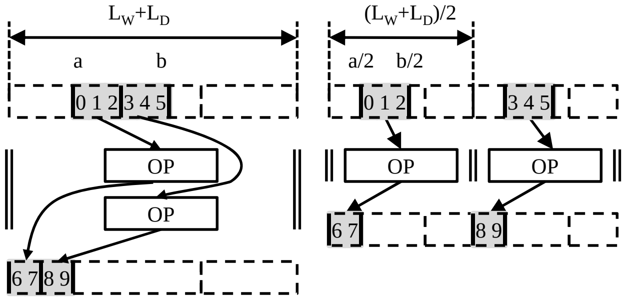
If the schedule fires an operator OP_i R_i times, these firings are independent and can be executed in parallel in a number of $\mathcal{C}$ threads. Therefore, we allow mapping each of the W steady state executions of the schedule to S $\mathcal{C}$ threads of the GPU. This effectively multiplies the available parallelism, and is essential in improving the GPU's utilization. Otherwise, the number of $\mathcal{C}$ threads utilized would be limited by the size of the SM memory. Accordingly, we split the WS buffer of a steady state execution into equal sections associated to each $\mathcal{C}$ thread. If an operator fires for less than S times, then it will be assigned to some of the threads, while the remaining threads will be idle, without any additional performance penalty. Operators firing more than S times will be executed several times by each $\mathcal{C}$ thread. Such a mapping is valid $\forall S$ such that $\forall i, \gcd(R_i, S) = \min(R_i, S)$.

Another important feature of SM memory is that it is banked and therefore supports parallel accesses, provided they do not go to the same bank. If in lockstep, all the $\mathcal{C}$ threads in a warp are accessing the SM memory, *and* the accesses are all to distinct banks, then the hardware will *coalesce* the accesses into a parallel access [5]. We can arrange for the accesses to the SM memory to be coalesced as follows. The WS and DB buffers are stored in a contiguous area of SM memory. Since the number of banks is a power of 2 (typically 16), to enforce coalescing, we just need to ensure that $L_W + L_D$ is a odd number. If the gap between consecutive WS buffers is an odd number p, then any WS offset in thread i and thread $i + j$, $\forall i, \forall j < 2^b$ is separated by a distance $j \cdot p$ which does not divide by 2^b, thereby ensuring that all banks are used. Therefore, the total number of parallel executions, W, that can fit a memory of size L_{SM} is

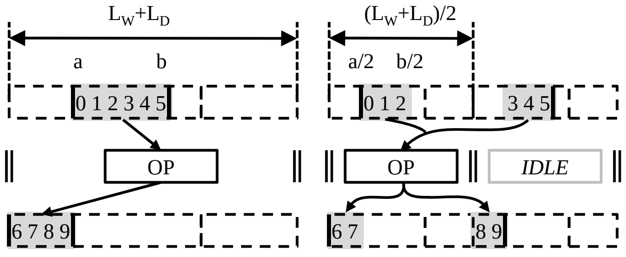
$$W \leq L_{SM}/\Lambda$$

where Λ is the buffer requirement for a single stream schedule execution, $\Lambda = 2 \cdot \lfloor \frac{L_W + L_D}{2} \rfloor + 1$.

Figure 3a compares the execution of an operator OP, scheduled to fire two times in a single thread with that of distributing



a) Scattering filter firings in GPU threads: S = 1 (left) and S = 2 (right)



b) Single filter firing with scattered data: S = 1 (left) and S = 2 (right)

Fig. 3. Buffer transformation examples.

it among two parallel threads. A region $[a, b]$ from a buffer of length $L_W + L_D$ will be subdivided into a set of smaller buffers, each of length $\frac{L_W + L_D}{S}$. The elements in the buffer are redistributed in sequence among the smaller buffers, filling the buffer of one thread before continuing to the next. By doing so, if the stream operator OP fires $f \cdot S$ times in the schedule, its firings can be distributed among S threads, in parallel, each thread handling f firings using data from its properly aligned section of the WS buffer. We avoid most of the additional synchronization overhead for this scheme by taking advantage of the lockstep nature of the threads in the same warp. The original buffers also need to be aligned to a multiple of S elements. Alternatively, Figure 3b shows how we execute a single firing of operator OP, when two threads are implemented. By means of a conditional, we simply disable the execution in the second thread. The operator running in the first thread can access elements from both WS buffers, and the same coalescing properties are maintained among the active threads in a warp.

C. Stream graph orchestration

A complete example of our method to orchestrate parallel executions of the steady state schedule, each onto multiple $\mathcal{C}$ threads ($S = 2$ in this example), is shown in Figure 4. The stream graph in the shaded box on the left is automatically translated to the execution scheme to its right. Whenever possible, operator firings are handled by parallel $\mathcal{C}$ threads. Each thread is allocated a buffer size of half the total WS size, precomputed for the entire steady state schedule. The WS buffer stores the intermediate results for future operator firings. For clarity, we shade those parts of the WS buffer in

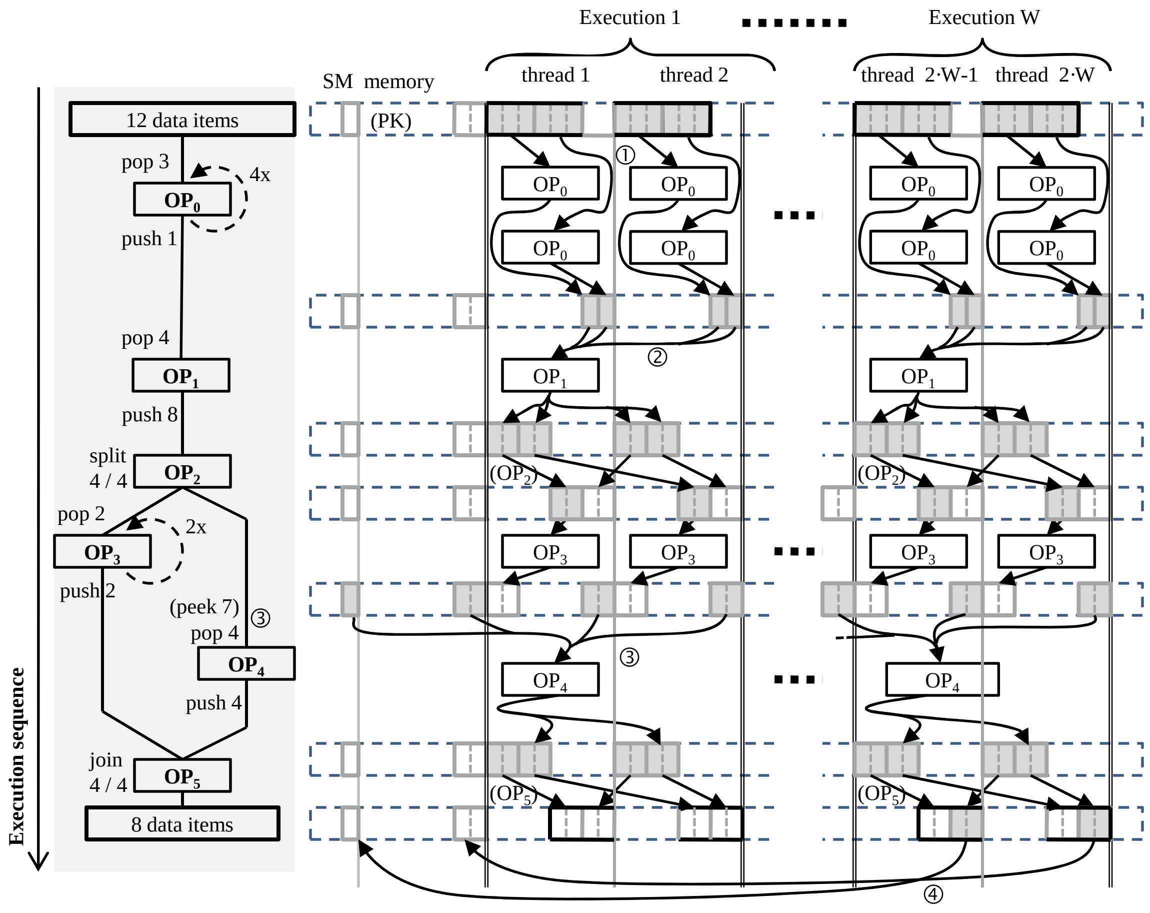


Fig. 4. Example of the orchestration for a single group iteration. Two $\mathcal{C}$ threads are assigned for each of the W parallel executions of the stream graph.

the figure used as input by the current operator firing. Also, we do not include in this illustration the DB buffer.

In this example, the 12 input items consumed by the stream graph during each execution of the steady state schedule are distributed among the SM memory buffers of the two $\mathcal{C}$ threads corresponding to each execution. Because OP_0 pushes only one element but OP_1 pops four elements, the schedule will consist of four firings of OP_0 for each firing of OP_1. We distribute OP_0's firings among the two threads, two in each thread ①. The outputs of OP_0's firings are written back to the WS buffer of both threads using a similar layout.

OP_1 needs four elements in a single firing, executed in the first thread, so it requires access to both its own WS buffer and the adjacent thread's WS buffer, both in the SM memory ②. To avoid the run-time overheads, we generate precomputed tables that translate the 0-based consecutive indexes of pop and push operations into relative offsets to the beginning of the allocated WS buffer. These relative offsets specify access ranges beyond the WS buffer limit of the current thread, and thus enable the fetching of data produced by adjacent threads that cooperate for the same schedule execution.

The output of OP_1 is the input of the splitter OP_2. The splitter divides the eight data items into two distinct buffers of four items. As necessitated by our mapping, each of these output buffers also need to be distributed in the WS buffers of both threads. Therefore, the splitter operator we generate distributes consecutive groups of two elements between the two threads' WS buffers. The execution of OP_3 and OP_4 is serialized in the steady state schedule. Each firing of OP_3 utilizes the set composed of the first two elements from each WS buffer, and runs in one of the two GPU threads. OP_3 does not utilize the second set of elements generated by OP_2.

Support for peeking: OP_4 is a *peeking* operator. In this example, OP_2 is required to push seven elements to the input of OP_4, before the latter can be fired ③. However, only the first four elements produced will be consumed. Therefore, the semantics of peeking requires preceding operators in the schedule to generate more data, which will be only inspected, but not consumed. In the current execution, OP_2 only generates four elements for OP_4. OP_4 must obtain the other three from another execution's OP_2 either in the current or the previous group iteration. We handle peeking in our scheme by shifting the buffer reference of the peeking filter's input into the previous execution's WS buffer. Intuitively, the first accessed elements in the sequence, which are those popped, were generated during a previous execution of the steady state graph, while the most recent ones, generated by the current steady state execution, are only peeked. Our precomputed tables take into account the popping/peeking requirements, and may contain negative relative offsets at the beginning of the sequence so that peeking filters can access elements in of the previous execution's WS buffer.

We precompute all the necessary offset tables on the host CPU and we preload them in the constant memory. We need to precompute such tables for each type of operator input / output rate. For example, for a filter having a pop rate p and a peek rate e, the input table T has e elements computed as follows: $\forall i \in [0, e], T_i = \frac{(p-e+i)\cdot S}{p} \cdot \text{size}(\Lambda) + \frac{(p-e+i)\cdot S \bmod p}{S}$. The first term determines the WS buffer to access and adjusts the offset by the relative offset of that buffer with respect to the current buffer. The second term specifies the relative position inside the WS. Integer division returns the lower integer as the result, while the $\bmod$ returns only positive values. As the constant memory is cached and the practical number of tables is small, this indirection has lower overhead than computing the values at runtime.

To support this peeking scheme in all parallel executions, we need to reserve a section (named 'PK' in Figure 4) at the beginning of the SM memory, where we copy the content of the previous input buffer of the peeking operators belonging to the last executions of the previous group iteration. This is necessary to expose the additional elements required by the first parallel $\mathcal{C}$ threads of the current group iteration. Suppose the current group iteration is j. OP_4 of execution $k, k > 1$ of group iteration j will obtain the three additional elements from execution $k-1$ of group iteration j, as they were written as a result of OP_2. The situation for execution 1 is special. OP_4 of execution 1 of group iteration j will have to get them from execution W of group iteration $(j-1)$ via the PK area. In this example, we copy these last three elements as the last step of the schedule execution in group iteration $(j-1)$, because OP_4's input buffer is not reused later ④.

To ensure access consistency to elements from adjacent executions, we introduce additional synchronization among the $\mathcal{C}$ threads before firing each peeking filter. This guarantees that the $\mathcal{C}$ threads belonging to different warps have completed execution of predecessor operators, and have produced all the necessary input data. Because we need to synchronize only $\mathcal{C}$ threads and not interfere with the $\mathcal{M}$ threads, we cannot use the SM thread synchronization primitives. We propose a simple workaround barrier that takes advantage of the lockstep execution within a warp. A thread representative is appointed for each $\mathcal{C}$ warp. This owns and increments a counter residing in SM memory when it reaches a synchronization point. Afterwards it repeatedly checks if its counter has a value smaller or equal to the other appointed threads' counters. If not, it waits. To avoid busy waiting, we force the hardware scheduler to run other warps by accessing a global memory location marked as volatile. Because all the threads in such a warp are in lockstep, this reduces the workload required, while holding all the warp's threads synchronized.

In addition, stream graphs containing peeking operators, need a special initialization schedule before the steady state groups can begin. This is necessary to initialize the buffers accessed during peeking. Otherwise, for example, OP_4 of execution 1 of very first group iteration would never have the additional three elements needed to be fired up. We can determine statically the number of required initialization iterations, and our scheme coordinates the GPU to execute an additional number of group iterations of the steady state for which it ignores the final outputs, but it updates all the intermediate values in the WS buffers, thus initializing them. We statically determine the correct offset in the input stream which enables the first group iteration of the steady state to fully utilize all the $\mathcal{C}$ threads after buffer initialization.

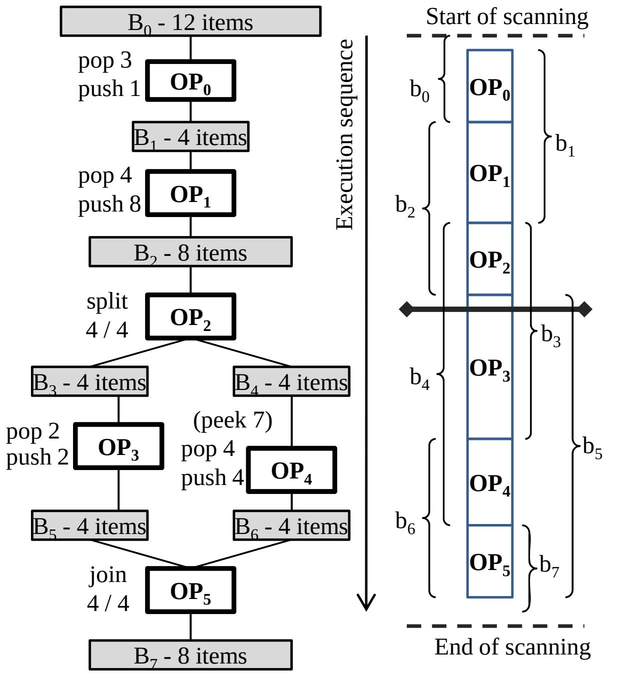
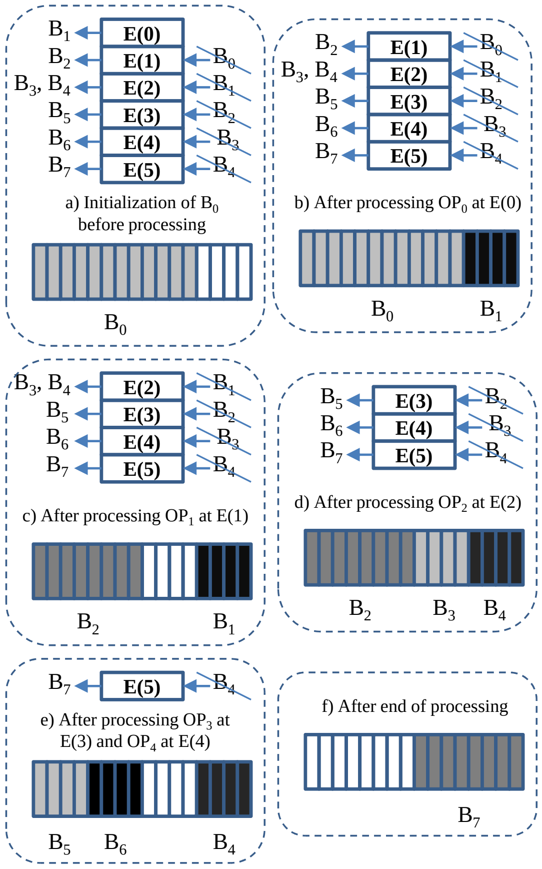


Fig. 5. Liveness analysis and lower bound of required buffer computation.

V. WS BUFFER LAYOUT

The size of the WS buffer stored in SM memory has direct impact on the performance of our mapping. The amount of SM memory is small, and thus a compact WS will enable a larger number of parallel stream executions. We present a simple algorithm that provides a near-optimal WS buffer layout for any stream graph. We first identify a lower bound on the WS buffer size. Next, using a simple yet efficient heuristic, we perform buffer allocation, slightly increasing the WS buffer size, if necessary, to accommodate this layout.

Figure 5 revisits the stream graph example in Section IV, showing the buffer requirements for each operator. Filters have a single input and output buffer each, while splitters and joiners transfer data from and to multiple buffers. An operator can be fired, if, and only if, its input/output buffers are in memory before and after its firing. Each buffer is written and read only once. Therefore, our mapping needs to arrange the layout of the buffers to prevent overwriting buffers before the data they contained is used.



a) Initialization of B0 before processing
b) After processing OP0 at E(0)
c) After processing OP1 at E(1)
d) After processing OP2 at E(2)
e) After processing OP3 at E(3) and OP4 at E(4)
f) After end of processing

Fig. 6. Buffer allocation example.

Let B_k be the buffer between two operators. In particular, let it be the output buffer of operator OP_i and the input buffer of operator OP_j. We define the *liveness* interval of B_k as the interval $b_k = [E(i), E(j)]$, where $E(n)$ is the position of operator OP_n in the execution schedule **E**. In Figure 5 we show the liveness interval of each buffer in the stream graph. For example, the liveness interval b_4 begins before the firing of OP_2 and ends after the firing of OP_4.

Based on the liveness intervals, we can compute the lower bound of the WS buffer size for the entire stream graph as follows. We scan linearly the execution of the N operators in the steady state (as shown in Figure 5) and we determine the minimum WS buffer size as $L_B = \max_{k\in[E(0),E(N)]}(\sum_{\forall n,k\in b_n} \mathrm{size}(B_n))$.

This lower bound is the minimum WS buffer size that can store all the necessary buffers during the entire execution of the steady state of the stream graph. The computation of this lower bound does not take into account the memory fragmentation caused by the constraint that each buffer must be a single, contiguous block of memory. We do not allow buffer relocation. Instead, we may have to increase the WS buffer size slightly in order to accommodate this constraint.

Given the lower bound of the size of the steady state WS buffer, we now describe a heuristic that uses this bound as a starting point and allocates buffers for each operator. Figure 6 walks through the allocation algorithm for the above mentioned stream graph and uses the lower bound of WS buffer size identified as $L_B = \mathrm{size}(B_5 \cup B_6 \cup B_7) = 16$). Initially, we allocate the input B_0 in the WS buffer (a). After $E(0)$, B_1 is placed into the SM memory (b). When processing $E(1)$, according to the liveness analysis, the space utilized by B_0 can be reused for B_2 (c). Next, splitter OP_2 will also have its output allocated (d). After the analysis of $E(3)$ and $E(4)$ (e), the joiner OP_5 has all its input allocated, and its output is allocated at $E(5)$, completing the steady state schedule analysis (f).

Algorithm 1 Buffer allocation algorithm

```
Require: steady state schedule $E$; number of operators $N$;
    lower bound of WS buffer $L_B$
Ensure: $allocation$ for each buffer in the WS, feasible WS
    buffer size $L_W$;
 1: $L_W \leftarrow L_B$;
 2: for i = 0 to N − 1 do
 3:     for each $b_j = [\dots, E(i-1)]$ do
 4:         deallocate($b_j$);
 5:     update_availability($L_W$);
 6:     for each $b_j = [E(i), \dots]$ do
 7:         if (find_next_slot($b_j$)) then
 8:             allocate($b_j$);
 9:             update_availability($L_W$);
10:         else
11:             extend($L_W$);
12:         record_allocation($b_j$, $allocation$);
13: return $allocation$, $L_W$;
```

Algorithm 1 summarizes our buffer allocation strategy. To allocate buffers for each ready operator in the execution schedule, we first update the availability of the WS buffer, deallocating all the buffers for which liveness has ended (line 3). The memory for the deallocated buffers will become available, and is combined to form large contiguous blocks of available memory. For each buffer that becomes live at this step, we search for an available memory slot (line 7, though not shown in detail) using a simple heuristic: we start from the last successful allocation, and try to find the nearest slot that will fit the current allocation request. The intuition is that neighboring buffers tend to expire together or close to one another, thereby increasing the likelihood of large chunks of contiguous free slots. If we are still unable to find a suitable memory slot, we extend the current WS buffer to fit the current buffer (line 11). Note that if there is some available memory at the rear of the WS buffer, we only need to extend its size by the difference to accommodate the new buffer. Finally, we return the allocated configuration and the final WS buffer size L_W (line 13).

We apply several constraints to the algorithm described above. We enforce a buffer alignment that is equal to the split factor S, such that we enable the splitting mechanism described in Section IV. Furthermore, peeking operators can not overlap their input buffers as the data is saved at the same offset in the PK section of the SM memory. So if two peeking filters overlap in their input buffers, their PK buffers will also overlap. However, the allocation in the PK section for peeking operators never expires, and so will result in a conflict. Therefore, while analyzing the stream graph, we maintain a set of 'visited' peeking operators and their buffer requirements, and we avoid allocating another peeking operator in the same memory segment. However, this issue does not occur between a peeking operator and a non-peeking one as the latter does not have a persistent presence in memory.

We also introduce a special optimization for duplicate splitters. These splitters are a special type of splitters that generate multiple identical output buffers from a single input buffer. To prevent expensive data movement, we simply extend the liveness of its input buffer until the last use of the splitter's original outputs.

VI. Characterization of Mapping on Different GPUs

In this section, we characterize the parameters used for our mapping strategy. We have used benchmarks that are packaged along with the StreamIt compiler [18]. The three mapping parameters that determine the execution time were defined in Section IV, namely,

- W, the number of parallel stream schedule executions;
- S, the number of $\mathcal{C}$ threads per execution;
- F, the number of $\mathcal{M}$ threads that transfer data between global and SM memory.

We varied the parameters of our mapping in order to better understand their impact on performance. We also revisited the idea that the standard approach taken in hiding the latency of global memory, which is to maximize the number of $\mathcal{C}$ threads. However, our $\mathcal{C}$ threads will generally be available for execution, as their WS buffer is allocated in SM memory, and thus we only need to find the right balance of $\mathcal{C}$ and $\mathcal{M}$ threads that matches their workload. Scheduling is done at the granularity of a warp, so if $\mathcal{M}$ and $\mathcal{C}$ threads are in distinct warps, they will execute concurrently.

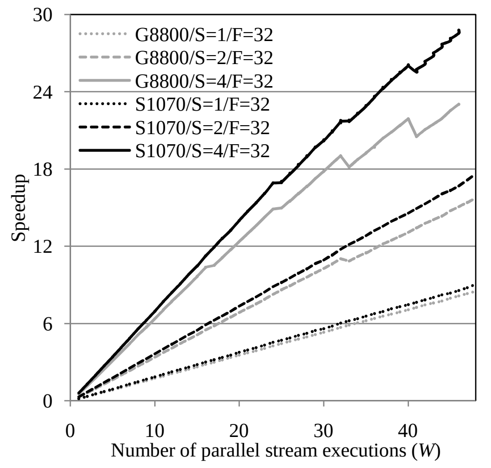
According to nVidia [5], hiding the latencies of the execution units requires 192 and 352 threads (6 / 11 warps) for devices of capability 1.x and 2.x, respectively. This assumes no global memory stalls and we will refer to this number as N_G. Therefore, we expect to see improvement in terms of execution time as long as we enable more $\mathcal{C}$ threads to run the stream graph schedule in parallel, until we reach N_G. As W is limited by the total size of the SM memory, we can increase the split factor S to enable more $\mathcal{C}$ threads.

Figure 7a characterizes the speedup we achieved based on the number of parallel stream executions for the FilterBank benchmark. We have selected a number of $\mathcal{M}$ threads ($F = 32$) high enough to sustain the transfer demands for the given design space. We then enumerated all the possible range of values for W and S. We measured the speedup achieved by the same benchmark configuration for two nVidia GPUs of capability 1.x, namely the G8800 and the Tesla S1070. The X- and Y-axis show the number of stream executions W, in each SM, and speedup, respectively. For each GPU type, different lines represent the speedup for different S values (number of $\mathcal{C}$ threads per steady state schedule execution). As expected, if the number of $\mathcal{C}$ threads increases, the speedup of the application increases accordingly. We define the speedup as the ratio of execution time of the application mapped to GPU compared to the execution time of a CPU (2.83 GHz Intel Xeon E5440) compilation. For the same number of iterations W, increasing S leads to higher speedup. The result also shows that the speedups on the S1070 are higher than those obtained on the G8800.

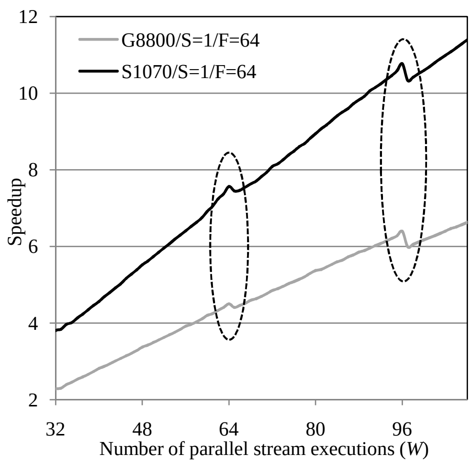
Does a higher number of $\mathcal{C}$ threads always guarantee higher speedup? Figure 7b shows an interesting result that higher number of $\mathcal{C}$ threads may hurt speedup. These anomalies can be explained by the correspondence of $\mathcal{C}$ threads to warps. If the number of $\mathcal{C}$ threads is a multiple of 32, warp occupancy will be at its highest, and only full warps are scheduled. On the other hand, if additional $\mathcal{C}$ threads are scheduled, the last warp is not only under-utilized but also occupies the same amount of GPU time allocated to the other warps. In Figure 7b, the speedup falls exactly at the above-mentioned points (because $S = 1$, the actual number of $\mathcal{C}$ threads is equal to the number of parallel stream executions). After a point, if the number of $\mathcal{C}$ threads continues to increase, the speedup gradually recovers due to increased warp occupancy.

As mentioned above, the number of $\mathcal{M}$ threads plays an important role when the $\mathcal{C}$ threads execute fast relative to the latency of global memory. Figure 8a shows the performance penalty when not enough $\mathcal{M}$ threads are scheduled for both the G8800 and S1070. We present experimental data for two different values of each GPU type: one in which the data demand of the $\mathcal{C}$ threads ($F = 32$) is not satisfied, and another in which it is ($F = 128$). After linearly increasing, the speedup corresponding to the smaller number of $\mathcal{M}$ threads reaches an upper bound, while the speedup corresponding to the higher number of $\mathcal{M}$ threads increases steadily on both GPUs. When the number of $\mathcal{C}$ threads is high enough, the data transferred by a small number of $\mathcal{M}$ threads is unable to keep up with the demand for data from the global memory. If the number of $\mathcal{M}$ threads increases correspondingly with demand, speedup increases nearly linear in terms of the number of $\mathcal{C}$ threads.

If the number of $\mathcal{M}$ threads is too high, performance (speedup) also degrades. Note that $\mathcal{M}$ threads compete for SM occupancy with $\mathcal{C}$ threads. All threads, irrespective of their type, are allocated an equal number of registers, and a higher SM occupancy leads to less registers available to each of the $\mathcal{C}$ threads. Therefore, performance may degrade due to register spilling as shown in Figure 8b. This effect is orthogonal to the one in Figure 7b, where no register spilling occurred. Our experiments show that the number of $\mathcal{M}$ threads typically required is 32 or 64. This result matches with the intuition that we do not need many $\mathcal{M}$ threads, because their task is only to match the demands of the W stream executions on each SM.

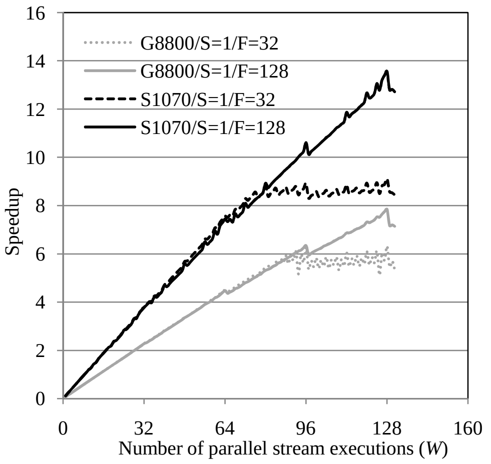


a) Design space characterization (FilterBank benchmark)

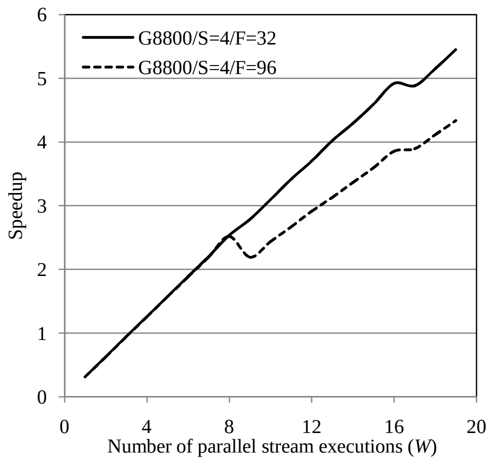


b) Speedup non-monotonicity (BitonicSort benchmark)

Fig. 7. Characterizing the design space.



a) Different designs varying *F* (BitonicSort benchmark)



b) Speedup penalty if *F* is too high (MatrixMult benchmark)

Fig. 8. The trade-offs for F, the number of $\mathcal{M}$ threads.

Heuristic equations for parameter selection: Based on these insights, we propose a set of equations to compute the correct number of $\mathcal{C}$ and $\mathcal{M}$ threads for any streaming application.

We first introduce an architectural constraint that requires that the number of $\mathcal{C}$ threads to be lower or equal to N_G because this number of threads fully utilizes the GPU in the absence of global memory stalls. $\mathcal{M}$ threads do not execute often, and we assume that they do not contribute to the total utilization. Thus, $W \cdot S \leq N_G$. We next include the constraint presented in Section IV-B and we derive the maximum number of parallel executions as a function of S:

$$W(S) = \min\left(\frac{N_G}{S}, \frac{L_{SM}}{\Lambda}\right)$$

The execution time $T(\mathbf{E}, S)$ of a group iteration depends on how the steady state schedule, $\mathbf{E}$, maps on the S $\mathcal{C}$ threads. Only operators fired iteratively in the schedule of the stream graph can be distributed and subsequently lower the execution time. Therefore, we analyse the execution schedule $\mathbf{E}_t^S$ for each thread t from the set of S threads associated with a stream graph execution. We obtain information about the estimated workload $\mathrm{WL}(p)$ of each operator OP_p from the StreamIt compiler. Putting these together, we get:

$$T(\mathbf{E}, S) = \max_{t<S}\left(\sum_{p \in \mathbf{E}_t^S} \mathrm{WL}(p)\right)$$

To maximize the speedup, i.e. $W(S)/T(\mathbf{E}, S)$, we need to determine S_m such that $\forall S_i \neq S_m, W(S_i)/T(\mathbf{E}, S_i) \leq W(S_m)/T(\mathbf{E}, S_m)$ which corresponds to $W' = W(S_m)$ executions. However, as suggested by Figure 7b, we need to ensure that packing the $W' \cdot S_m$ threads into warps does not leave the number of active threads in the last warp to be less than $(W' \cdot S_m)/16$. If the last warp is underutilized then we reduce W to $\lfloor \frac{W' \cdot S_m}{32} \rfloor \cdot 32$, otherwise use $W = W'$.

In addition, we analyse the ratio of parallel executions to $\mathcal{M}$ threads. This has to match the ratio between the run time of the parallel stream executions and their IO data set size L_D.

TABLE I
BENCHMARK CHARACTERIZATION

Benchmark	Description
Bitonic	Sorting algorithm for 8 float elements applying the bitonic algorithm
BitonicRec	The same as above, recursive method
DCT	Discrete Cosine Transform for a matrix of 8x8 float elements
DES	DES encryption algorithm, input 8 bytes, output as 16 hex digits
FFT	Fine grained FFT transform, with 32 inputs
FilterBank	Instantiates 4 filter banks for multirate signal processing
FMRadio	11-band equalizer radio
MatrixMult[3]	Blocked matrix multiplication algorithm for 4x4 matrices, split into blocks of 2x2

$$\frac{W}{F} = k \cdot \frac{T(\mathbf{E}, S)}{L_D}$$

where k is a GPU-dependent constant we derive experimentally. We round F to the next full warp value.

VII. EXPERIMENTS

Based on the heuristic presented above, we can efficiently select the number of $\mathcal{C}$ and $\mathcal{M}$ threads. In the following experimental results we shall compare the speedups between:

- the previous state of the art implementation [9] and our results;
- different nVidia architectures.

We start by comparing our mapping scheme with the results presented by a recent work [9], already described in Section II. We shall refer to this by the acronym 'UGT'. This work partitions the stream graph between SMs, and launches a large set of homogeneous parallel threads in each SM. Data transfers between SMs are done via the global memory.

We develop our mapping flow at the back-end of the StreamIt 2.1.1 compiler. As presented in Section IV, the output of our mapping can be compiled and run on different GPU architectures with the correct number of parameters as selected by the heuristic equation. In order to match the experimental setup of UGT, we ran one set of experiments on the nVidia G8800 with an old driver of release number 177.73. As a baseline, we use the same platform as UGT, namely, an Intel Xeon E5440 running at 2.83 GHz, with the executable obtained through the uniprocessor backend of StreamIt, and compiled using the '-O3' option of GCC 4.1.2.

We use the benchmarks found in the benchmark suite bundled with the StreamIt compiler. From the description found in their paper, we adjusted the benchmark parameters to be as close to those used by UGT as possible. A description of each benchmark is found in Table I.

[3]We were unable to deduce the configuration used by UGT for this benchmark based on their description. Instead, we use what is reported here.

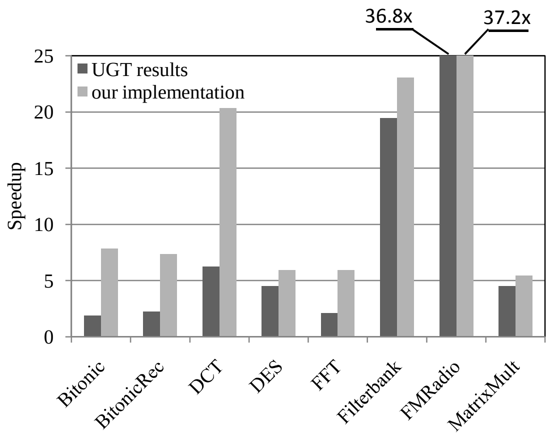


Fig. 9. The comparison between *UGT* and our solutions

TABLE II
BENCHMARK MAPPING PARAMETERS

Benchmark	Λ	G8800 / S1070			S2050		
	(words)	W	S	F	W	S	F
Bitonic	31	128	1	64	352	1	160
BitonicRec	31	128	1	64	352	1	192
DCT	193	20	4	32	62	4	96
DES	95	40	2	32	128	2	32
FFT	129	30	4	32	88	4	96
FilterBank	49	46	4	32	88	4	32
FMRadio	27	88	1	32	352	1	32
MatrixMult	209	19	4	32	56	4	96

Figure 9 shows the comparison between UGT and our approach. We consistently obtain better performance with our proposed mapping scheme. The speedup in the graph is the ratio of execution time on GPU to that on the CPU. For all 8 benchmarks, our solution executes faster than the UGT scheme, by as much as $4.2\times$. On average, ours is $2.8\times$ better than theirs. The smallest improvement is for FMRadio, but this is due to an opportunistic optimization that was introduced in the UGT implementation. Because the working set buffer of each iteration in this stream graph is relatively small, the entire work set for a large number of iterations was allocated in SM memory. This result actually confirms the direction taken by our mapping strategy.

In order to demonstrate the portability of our mapping scheme to different GPU architectures, we performed experiments on the nVidia G8800 (capability 1.0), Tesla S1070 (capability 1.3) and Tesla S2050 (capability 2.0). The results are shown in Figure 10 and Table II. For these experiments we have used the CUDA toolkit and driver version 3.1. We targeted a single GPU device on the multi-GPU Tesla platforms. For S2050, we enabled the extended 48 KB SM memory. This extended SM memory is mutually exclusive with a larger cache. In our case, as we can instruct the hardware which global memory data sets to essentially fetch, caching on demand would not have performed better.

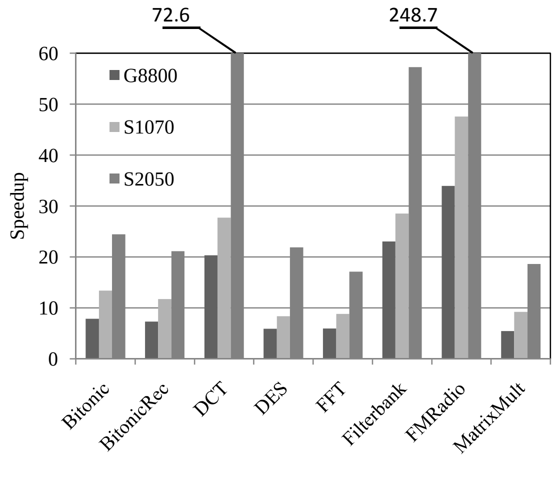


Fig. 10. The portability of the automated mapping scheme into G8800, S1070 and S2050 GPUs

The results show that the automated mapping scheme not only works on different GPU architectures, but it can also explore the advanced features of newer GPUs. On average, while mapping on the S1070 GPUs, speedups are $1.44\times$ better than on the G8800, and on the S2050 the performance is $2.62\times$ better than the S1070. We attribute this significant improvement to the additional processing cores and to the larger SM memory that allowed for a larger total number of parallel stream executions, i.e., W. It shows that our automated mapping scheme scales well with the current GPU development trend of increasing the number of processing cores and the size of the SM memory.

We have also attempted to port our mapping scheme onto AMD GPUs via OpenCL. We assigned $\mathcal{C}$ and $\mathcal{M}$ threads to different wavefronts (the AMD equivalent of warps), and experimented with an ATI HD5870 GPU board. However, while we did achieve linear speedups, our experiments showed that using $\mathcal{M}$ threads incurred up to 30% of overhead. We had expected the overhead to be significantly much lower. While AMD documentation does not fully disclose the wavefront scheduling algorithm, it is mentioned that a pair of wavefronts hides all the ALU execution latency [19]. Therefore, we suspect that if this pair of wavefronts contains only $\mathcal{C}$ threads, the scheduler disadvantages $\mathcal{M}$ threads, making them unable to load the data in time. The $\mathcal{M}$ threads became a liability instead. We would like to investigate this in the future.

VIII. Conclusions

We presented a novel and efficient scheme to execute stream graphs on GPUs that involves pipelining memory access threads that prefetch data from the off-chip memory to the on-chip memory, and compute threads that are disconnected from the off-chip memory. We support all the features of the StreamIt language, except stateful filters. Compared with previous mapping results of StreamIt to GPUs, our implementation always performs faster, by as much as $4.2\times$ better, on the same experimental setup.

Our performance characterization shows the non-trivial trade-off between memory access and compute threads, and we proposed a heuristic that assists in automatically selecting the best mapping parameters.

All the benchmarks we used could be implemented within a single partition. However, if the working set buffer of the steady state grows too large, our scheme supports the interleaved execution of multiple subgraphs, using the off-chip memory as intermediate storage. Even in this scenario, our approach still minimizes the transfer of data between on-chip and off-chip memory.

Orthogonal to our approach, performance may also be improved by the introduction of a buffer layout algorithm that is better than our current heuristic. This is supported by the observation that performance is inversely proportional to buffer size. We intend to explore this in our future work.

References

[1] M. I. Gordon and et. al., “A stream compiler for communication-exposed architectures,” in *ASPLOS ’02*, Oct 2002.
[2] J. Nickolls and W. J. Dally, “The GPU computing era,” *IEEE Micro*, vol. 30, pp. 56–69, 2010.
[3] J. D. Owens and et. al., “A survey of general-purpose computation on graphics hardware,” *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.
[4] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008.
[5] Nvidia cuda. Http://www.nvidia.com/object/cuda
[6] I. Buck and et. al., “Brook for GPUs: stream computing on graphics hardware,” in *SIGGRAPH ’04*. New York, NY, USA: ACM, 2004, pp. 777–786.
[7] M. I. Gordon, W. Thies, and S. Amarasinghe, “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs,” in *ASPLOS ’06*. New York, NY, USA: ACM, 2006, pp. 151–162.
[8] M. Kudlur and S. Mahlke, “Orchestrating the execution of stream programs on multicore platforms,” in *PLDI ’08*, 2008, pp. 114–124.
[9] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, “Software pipelined execution of stream programs on GPUs,” in *CGO ’09*, 2009, pp. 200–209.
[10] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.
[11] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, “Optimization principles and application performance evaluation of a multithreaded gpu using cuda,” in *PPoPP ’08*, 2008, pp. 73–82.
[12] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne, “High performance comparison-based sorting algorithm on many-core GPUs,” Apr. 2010, pp. 1–10.
[13] Hpc project, par4all. HPC Project, Par4All.
[14] F. Bodin and S. Bihan, “Heterogeneous multicore parallel programming for graphics processing units,” *Sci. Program.*, vol. 17, no. 4, pp. 325–336, 2009.
[15] M. Wolfe, “Implementing the PGI accelerator model,” in *GPGPU ’10*, 2010, pp. 43–50.
[16] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, “Cache aware optimization of stream programs,” *SIGPLAN Not.*, vol. 40, no. 7, pp. 115–126, 2005.
[17] L. Li, H. Feng, and J. Xue, “Compiler-directed scratchpad memory management via graph coloring,” *ACM Trans. Archit. Code Optim.*, vol. 6, no. 3, pp. 1–17, 2009.
[18] Streamit benchmarks. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.
[19] ATI stream computing programming guide. http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf.