Automatic Mapping of Stream Programs on Multicore Architectures

Pablo de Oliveira Castro, Stéphane Louise, Denis Barthou

▶ **To cite this version:**

Pablo de Oliveira Castro, Stéphane Louise, Denis Barthou. Automatic Mapping of Stream Programs on Multicore Architectures. International Workshop on Compilers for Parallel Computers, Jul 2010, Vienna, Austria. hal-00551680

HAL Id: hal-00551680

https://hal.science/hal-00551680v1

Submitted on 28 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automatic Mapping of Stream Programs on Multicore Architectures.

Pablo de Oliveira Castro[1], Stéphane Louise[1], and Denis Barthou[2]

[1] CEA, LIST
[2] University of Bordeaux - Labri / INRIA

Abstract. Stream languages explicitly describe fork-join and pipeline parallelism, offering a powerful programming model for general multicore systems. This parallelism description can be exploited on hybrid architectures, *eg.* composed of Graphics Processing Units (GPUs) and general purpose multicore processors.
In this paper, we present a novel approach to optimize stream programs for hybrid architectures composed of GPU and multicore CPUs. The approach focuses on memory and communication performance bottlenecks for this kind of architecture. The initial task graph of the stream program is first transformed so as to reduce fork-join synchronization costs. The transformation is obtained through the application of a sequence of some optimizing elementary stream restructurations enabling communication efficient mappings. Then tasks are scheduled in a software pipeline and coarsened with a coarsening level adapted to their placement (CPU of GPU). Our experiments show the importance of both the synchronization cost reduction and of the coarsening step on performance, adapting the grain of parallelism to the CPUs and to the GPU.

1 Introduction

Modern multiprocessor architectures combine an increasing number of cores and mix general purpose cores with dedicated accelerators (such as Cell or GPU for instance). Programming for heterogeneous machines is difficult since it requires the expression of parallelism of different kinds and of different grains. The stream programming models [4][9][5] are particularly adapted to tackle this issue and expose task-, data- and pipeline parallelism. Stream programs are seen as a set of filters (or tasks) interconnected through buffers or FIFOs and as a set of stream reorganization nodes.

One of the most prominent performance constraints of concurrent real-time applications is throughput. Optimizing throughput of a stream program in general requires to find a good task partitioning among the heterogeneous computing units (see [17],[2]) and then an efficient parallel schedule to hide communication latencies whenever possible. One of the difficulties of both partitioning and scheduling steps is to take into account the numerous architectural mechanisms involved – data flows between CPU and GPU correspond to communications

while data flows between cores of a same processor correspond to data transfers in the memory hierarchy.

In this paper, we present a novel approach to optimize stream programs for hybrid architectures composed of GPU and multicore CPUs. The approach focuses on memory and communication performance bottlenecks for this kind of architecture. The initial stream graph is first transformed by a sequence of elementary restructurations. The guided beam-search method applied aims to reduce fork-join synchronization costs. We show that the heuristic proposed to drive these restructurations obtains similar results to those obtained by an exhaustive search. The tasks are then partitioned between CPU cores and GPU using some existing partitioner, taking into account profiling information for a workload balanced partitioning. A new scheduling technique is finally proposed to coarsen tasks of each partition in order to adapt to cache sizes and communication constraints of CPUs and GPU. Our experiments show the importance of both the synchronization cost reduction and of the coarsening step on performance, adapting the grain of parallelism to the CPUs and to the GPU.

2 Background and Motivating Example

2.1 Stream Graphs

Our Stream Graph formalism, very close to StreamIt[4], describes a program using a synchronous data flow graph[8] where nodes are actors that are fired periodically and edges represent communication channels. Unlike StreamIt that imposes a serie-parallel structure on the stream graph, nodes can be composed freely in our model.

Source (**I**) and **Sink** (**O**) nodes model respectively the program inputs and outputs. The source produces a stream of inputs elements, while the sink consumes all the elements it receives. A source producing always the same element is a *constant* source (**C**). If the elements in a sink are never observed, it is a *trash* sink (**T**).

Functions in the imperative programming paradigm are replaced by **filter** nodes $\mathbf{F}(\mathbf{c_1}, \mathbf{p_1})$. Each filter has one input and one output, and an associated pure (with no internal state) function f. Each time there are at least c_1 elements on the input, the filter is fired: the function f consumes the c_1 input elements and produces p_1 elements on the output.

Another category of nodes dispatch and combine streams of data from multiple filters, routing data streams through the program and reorganizing the order of elements within a stream.

Join round-robin $\mathbf{J}(\mathbf{c_1} \ldots \mathbf{c_n})$: A join round-robin has n inputs and one output. Each time it is fired it consumes c_i elements on every i^{th} input, and concatenates the consumed elements on its output.

Split round-robin $\mathbf{S}(\mathbf{p_1} \ldots \mathbf{p_m})$: A split round-robin has m outputs and one input. A split consumes $\sum_i p_i$ elements on its input and dispatches them on the outputs (the first p_1 elements are pushed to the first output, then p_2 elements are pushed to the second, etc.).

Duplicate D(m) has one input and m outputs. Each time this node is fired, it takes one element on the input and writes it to every output, duplicating its input m times.

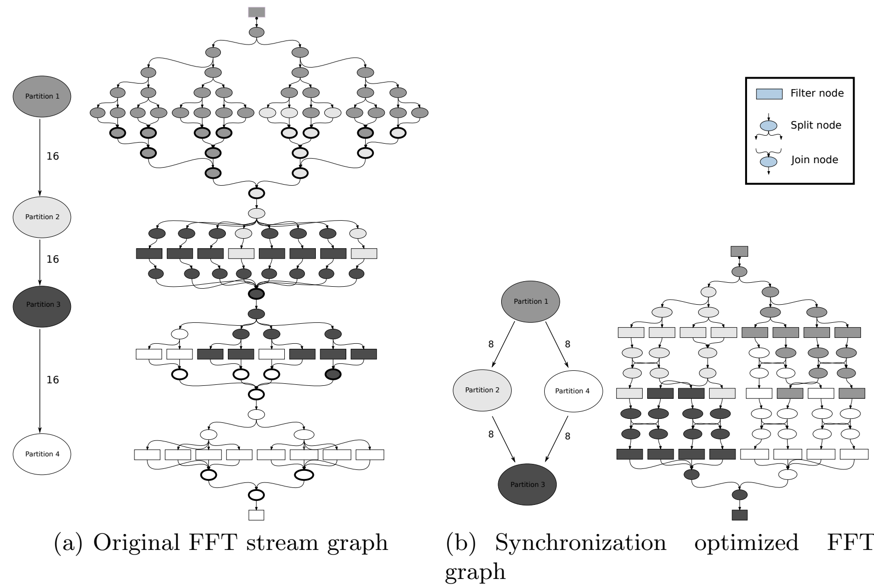
We can schedule an SDFG in bounded memory if it has no deadlocks and is consistent. As proved in [8] a consistent SDFG admits a repetition vector $\boldsymbol{q_G} = [q_1, q_2, \ldots, q_{N_G}]$ where q_N is the repetition number of node N. A schedule where each actor N is fired q_N times is called a *steady-state* schedule. Such a schedule is rate matched: for every pair of actors (U, V) connected by an edge E, the number of elements produced by U on E is equal to the number of elements consumed by V on E, during a steady-state execution. This number of elements is noted $\beta(E) = prod(U) \times q_U = cons(V) \times q_V$. Once a steady-state schedule is found, we may *coarsen* it. That is to say, replace each actor N by a coarsened actor which fires $coarse \times q_N$ times per schedule tick. This allows to adjust the number of elements processed by each actor during a schedule tick. As described in next section, it plays an important role in a stream program performance.

2.2 Motivating Example

Our target architecture is composed of a Nehalem QuadCore (Xeon W3520 at 2.67GHz) and a NVIDIA Quadro FX 580 GPU with 4 Streaming Multiprocessors. When mapping a Stream Program to this architecture, we can exploit two kinds of parallelism: Spatial parallelism, by distributing the nodes of the graph among the available cores, and temporal parallelism, by software pipelining the successive executions of the nodes. In this context, performance is going to be determined by three main factors: (i) The time taken to complete one schedule tick by each processor. (ii) The time taken to transfer data from the CPU and GPU in one schedule step. (iii) The time taken to transfer data from one CPU to another CPU, which depends on whether the data is in the L3 cache. The final performance will be determined by the maximum cost among these interdependent factors, which need to be balanced.

Partitioning We want to evenly distribute the work cost of the nodes among the cores. This is done by partitioning the graph in as many partitions as cores (CPUs + GPU). The time to execute a node in one steady-state execution will be refered as the work cost of the node. We consider as well the communication cost between partitions, defined as the number of elements that are streamed in one steady-state execution between two partitions.

Our objective is a workload balanced partitioning with the smallest possible communication cost between partitions. Consider the FFT Butterfly Stream graph in figure 1(a), using the METIS[6] graph partitioner, we have found four work balanced partitions. The partitioning found is laid in a vertical fashion. The communication cost of this partitioning is high, METIS could not find a better partitioning because it cannot cut through the the synchronization nodes with bold edges in figure 1(a). To avoid this problem, we propose to do a synchronization removal step before doing the actual partitioning. By applying a set of legal



(a) Original FFT stream graph

(b) Synchronization optimized FFT graph

Fig. 1. Partitioning a finegrained FFT stream graph: figure (a) is the original version, the first stage only composed by Split and Join nodes is a shuffling stage that puts the elements in the order required by the FFT Butterfly; figure (b) is the transformed version, the butterfly pattern is now apparent in the graph. Both version share, as expected, the same number of filters and the same total throughput. The figures to the left, represent the selected partitioning for each version with the volume of communication per steady-state between each pair of partitions.

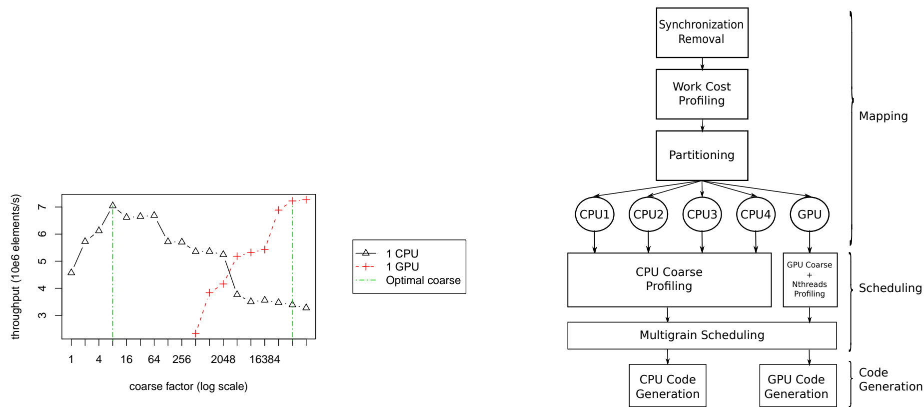


Fig. 2. (a) Impact of the coarsening on the FFT throughput on one CPU and one GPU; (b) Compiler chain overview.

graph transformations, we rewrite the original SDFG into a new version. When applied to the FFT graph, this technique breaks the synchronization nodes into smaller constituents, producing the optimized graph in figure 1(b) which offers more liberty to the partitioner to find a communication efficient layout. As you can see the partitioning layout has decreased the inter-partition communication cost from 48 elements per steady-state to 32 elements per steady-state.

Scheduling In the graph in figure 2(a), we measured the throughput of the FFT stream graph on one CPU core and one GPU core with different levels of coarsening. The coarsening level is the number of steady-state executions per schedule tick. We notice that the CPU version is most efficient with a coarsening level of 2^3, while the GPU version is most efficient with a coarsening level of 2^{17}. We attribute this difference in optimal coarsening levels to two effects. On the CPU, when the coarsening is small, the memory used by one schedule tick is small enough to fit in the cache. Successive executions in the software pipeline will take advantage of cache reuse. On the other hand, when the GPU works with small coarse sizes, few elements are packed in each DMA tranfer, so the high initial DMA cost dominates the pipeline. When using big coarse sizes, the DMA cost is amortized by the large number of elements sent, and covered by the increased computation time. To accomodate the requirements of both CPU cores and GPU we propose a *multigrain scheduling*, that combines the two coarsening levels as described in section 5.

3 Compilation overview

The main steps of our compiler are outlined in figure 2(b). The process starts with the *Syncronization removal*: in this step the transformations described in section 4.1 are used to reduce the number of synchronization nodes in the graph, potentially improving communication costs of workload balanced partitioning.

Then the actual graph partitionning is achieved, mapping each stream actor to a computation unit (CPU core or GPU). We first measure each node work cost by doing a profile run on the CPU (*Work Cost profiling*) and GPU; then load-balance the work costs among the cores with the minimal inter-core communication cost. We measure the optimal coarse factors C_{cpu} and C_{gpu} by running each partition with different coarsening in the range $[2, 4, \ldots, 2^{20}]$. Given our limited number of benchmarks, it would seem that the optimum coarse size depends on the ratio between the computation time and the communication cost. Therefore we believe that these values could be automatically derived from the node work, node communication cost and frequence and communication speed in the GPU/CPU. This will be the object of future work.

We finally compute the multigrain schedule as explained in section 5. Each CPU partition is compiled as a single thread that runs either on the CPU or on the GPU. Each thread has main loop that calls the node work functions in the order prescribed by the static multigrain schedule. Transfers between CPU and GPU are scheduled by a special CPU host thread which allocates the pinned

memory necessary for asynchronous DMA transfers. A shuffling operation is done on transfers between CPU and GPU, so the GPU can access data efficiently in a coalesced fashion[11].

For each thread a C or CUDA file is produced, which is compiled using `gcc-4.3` or `nvcc-2.3` and linked. The actual code compilation is similar to the Stream Graph Modulo Scheduling (SGMS) code generation described in [7].

4 Synchronization aware partitioning

Many task partitioning methods have been proposed in the literature. In the context of Stream Graph, optimal ILP based solutions[17], dynamic programming heuristic [14] and iterative graph partitioning solutions [2] have been presented. In this paper we do not introduce a new partitioner for SDF graphs, but concentrate instead on proposing a guided graph restructuration enabling communication effective partitioning. Partitioners considered are assumed to find partitions that have balanced workload. Moreover, for equivalent solutions, the partioning found minimizes inter-partition communication cost. For the evaluation of the workload, We assume that Split and Join node latencies depend linearly on the number of elements on their output (resp. input).

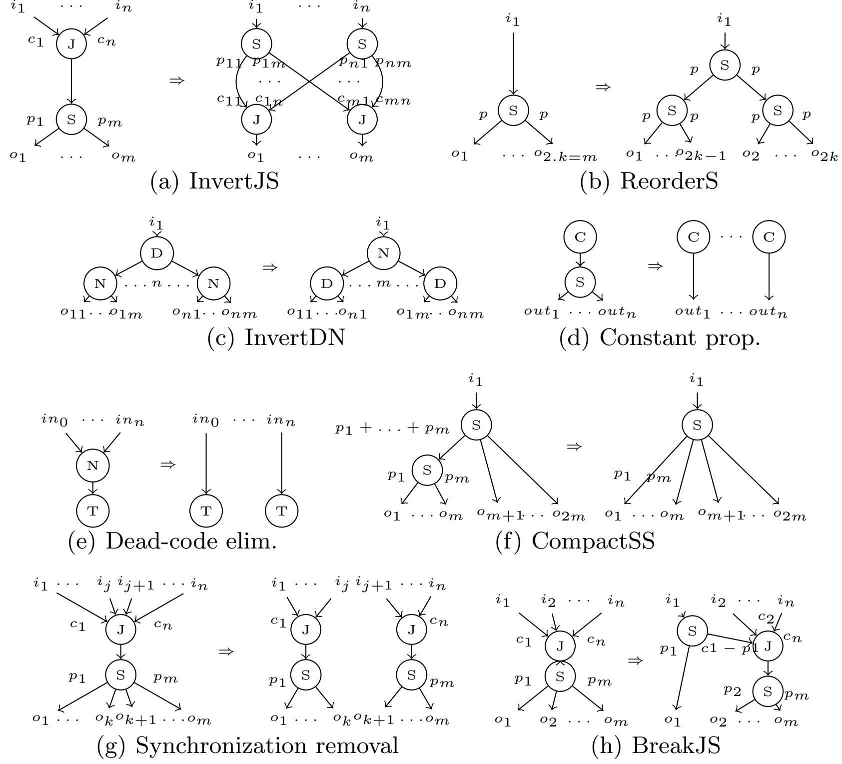
4.1 Transformations

We use all the transformations proposed in [12] (except SplitF). We represent some of these transformations in figure 3. These transformations can be separated in three groups according to their effect:

- **Node removal** *(RemoveJS / RemoveSJ / RemoveD / CompactSS / CompactDD / CompactJJ)* these transformations remove nodes which composed effect is the identity.
- **Synchronization removal** *(Constant propagation / Dead code elimination / BreakJS / Synchronization Removal* These transformations, break synchronization points inside a communication pattern, usually by decomposing it into its smaller constituents.
- **Restructuring** *InvertDN / InvertJS / ReorderS / ReorderJ* These transformations restructure communication patterns, alone they do not improve the communication metric, but they can rewrite the graph and trigger some of the previous transformations.

4.2 Graph Restructuration for Synchronization Removal

The throughput corresponds to the number of elements produced by the graph per time unit. When the graph is partitioned among different computing units, the throughput is determined by the execution time of each partition and the communications resulting from the partitioning. Even if the workload is well balanced among the different units, communication time may hinder performance.



(a) InvertJS (b) ReorderS (c) InvertDN (d) Constant prop. (e) Dead-code elim. (f) CompactSS (g) Synchronization removal (h) BreakJS

Fig. 3. Set of transformations considered. Each transformation is defined by a graph rewriting rule. Node N is a wildcard for any arity compatible node.

Communications between computing units on a hybrid architecture correspond to different hardware mechanisms. Communication time between CPU cores is drastically reduced if the data to transfer fits into a cache shared by the two communicating cores. For CPU-GPU communication, communication time increases with the number of elements to transfer. In both cases, reducing the number of elements required by a computing unit (one core or one GPU) can only reduce the communication time. And reducing the communication time between the different partitions will increase the throughput for communication-bound graphs. The method we propose restructure the stream graph so as to remove potential communication bottlenecks.

Consider a workload balanced partitioning $\mathcal{P}$ of a stream graph G. The total number of input elements required for one time step by all partitions is defined by:

$$incomm(\mathcal{P}, G) = \sum_{Q \in \mathcal{P}} \sum_{E \in in(Q)} \beta(E)$$

where $in(Q)$ is the set of incoming edges for partition Q. $incomm(\mathcal{P}, G)$ evaluates the amount of communications per time step, according to the partitioning chosen. The goal is to transform the graph into G' so that the amount of communication required in G' is lower than in G. The following lemma shows that this is ensured by the transformations considered:

Lemma 1. *For any input stream graph G, consider G' a graph obtained after any number of transformations described in 4.1. Let $\mathcal{P}$ be any balanced workload partitioning of G and $\mathcal{P}'$ any balanced workload partitioning of G', then*

$$incomm(\mathcal{P}', G') \leq incomm(\mathcal{P}, G)$$

The proof of this result is obtained by showing for each one of the transformations in [12] (except for SplitF, for which this result does not apply) that given any balanced partitionning in the original match subgraph, it is possible to find a new balanced partitionning on the replacement subgraph which does not increase *incomm*.

We have shown that our transformations can only improve the communication cost of the partitionning but we do not know which transformation sequence is optimal. If we could evaluate the best mapping for each explored variant we could easily find the best sequence. Sadly, the cost of finding the best mapping for each variant would be prohibitive. Therefore to chose among the many possible sequences of transformations, none of them increasing the communication cost, we propose a new metric (that does not depend on the partitionning):

$$mcost(G) = mean(\{commcost(N) \quad : N \in G \quad \text{iff} \quad |in(N)| \geq 2\}),$$

where $commcost(N) = \sum_{E \in in(N)} \beta(E)$. The value $commcost(N)$ corresponds to the number of input elements of a node N. $mcost(G)$ is the mean value of input elements for nodes with an arity ≥ 2, therefore corresponding to Join nodes. These nodes are potential bottlenecks since they introduce a synchronization point between different branches. It can be shown that restructuring and node removal transformations preserve the value of *mcost* in the graph, while synchronization removal transformations reduce this value. As the latter correspond to transformations breaking useless dependences, they enable a wider range of partitioning, potentially reducing communication costs.

4.3 Exploration

In [12] the authors prove that by considering all the possible combinations of these transformation a very large but finite number of versions is generated. They propose an exhaustive exploration of these versions to reduce the memory of stream graphs.

In this paper we choose to explore this search space using a Beam Search greedy algorithm [10]. At each search step, Beam Search considers all the transformations that could be applied to the stream graph. Each one of the resulting candidates is evaluated with the metric. Beam Search orders the candidates according to *mcost*, and discards all the candidates except the first *beamsize* ones. The selected candidates, create new search branches that are recursively explored with the same algorithm, until no more transformations can be applied.

In our compilation framework we use *beamsize* = 2, and find candidates almost as good as an exhaustive search would. The table below show the percentage between the best *mcost* solution found by the 2-beam search and the optimal *mcost* solution found by an exhaustive search.

	FFT	DCT	MatMul	Bitonic
Percentage to the optimum	85.9%	100%	100%	89.7%

The exploration is particularly memory efficient because it never copies the graph when branching. Transformations are in-place applied and in-place reverted when backtracking. The 2-beamsearch never takes more than 3 minutes in a commodity desktop computer.

5 Multigrain Scheduling

We have seen in section 2 that the optimal coarsening for the CPU caches (C_{cpu}) is small whereas the optimal coarsening for the GPU bus (C_{gpu}) is big. An efficient way of scheduling SDFG is Stream Graph Modulo Scheduling (SGMS), introduced in [7]. In a SGMS schedule the production and consumption rates of the actors must be matched, we can only apply a single coarsening factor to an SGMS schedule. Therefore, if we were to use a single schedule, we would need to find a trade-off between the cache optimal coarse grain and the GPU bus optimal coarse grain. As shown in figure. 2(a), there is no optimal configuration that satisfies both requirements at the same time.

To avoid the trade-off we introduce the *multigrain* schedule that nests two SGMS schedules: the *outer schedule* and the *inner schedule*. The CPU nodes run inside the *inner schedule*, so they work on buffers of C_{cpu} size, small enough to stay in the cache taking advantage from cache reuse. We run GPU nodes on the *outer schedule* so they work on buffers of C_{gpu} size, big enough to pay for DMA costs. We ensure that the producer-consumer rate is the same at the boundaries of the two schedules, by executing C_{gpu}/C_{cpu} times the *inner schedule* for each *outer schedule* tick.

Multigrain stage assignment As in SGMS, each node is assigned a stage that decides its time of activation in the software pipeline. In Multigrain scheduling, each node possesses a couple (os, is), where os is its stage in the *outer schedule* and is the stage in the *inner schedule*. The stages are chosen in algorithm 1, which enforces two simple rules:

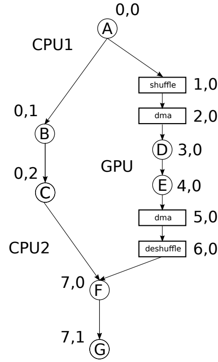
- (Rule A) Preservation of data dependences: data dependencies are satisfied at the *inner schedule* and *outer schedule* level; that is when an actor is fired all the elements it reads have already been produced in a previous stage.
- (Rule B) Overlapping DMA latencies with computation time: given two actors, one on the GPU, the other on the CPUs. We skip two free stages between the consumer stage and the producer stage. One of the free stages is reserved for the shuffling/deshuffling operation. The other stage is reserved for DMA transfer. This guarantees that the data needed by an actor is always shuffled and prefetched in the preceding schedule ticks.

In figure 4(a), we show a simple task graph and the stages selected by algorithm 1.

Algorithm 1 MultigrainScheduling(G)

input: SDF Graph G
output: SDF Graph G decorated with a couple (os, is)

1: $N.os \leftarrow 0$
2: **for all** $N \in$ TopologicalOrder(G) **do**
3:   **for all** $P \in$ Parents(N) **do**
4:     **if** $N \in CPU$ and $P \in CPU$ **then**
5:       $N.os \leftarrow max(N.os, P.os)$
6:     **else if** $N \in GPU$ and $P \in GPU$ **then**
7:       $N.os \leftarrow max(N.os, P.os + 1)$
8:     **else**
9:       $N.os \leftarrow max(N.os, P.os + 3)$
10:     **end if**
11:   **end for**
12:   $N.is \leftarrow 0$
13:   **for all** $P \in$ Parents(N) **do**
14:     **if** $N.os = P.os$ **then**
15:       $N.is \leftarrow max(N.is, P.is + 1)$
16:     **end if**
17:   **end for**
18: **end for**



(a) Stage assignment in a multigrain schedule

In0 inner-schedule

CPU1	CPU2
A0,0	
A0,1 B0,0	C0,0
A0,2 B0,1	C0,1
A0,3 B0,2	C0,2
B0,3	C0,3

In'0 inner-schedule

CPU2	
F0,0	
F0,1	G0,0
F0,2	G0,1
F0,3	G0,2
	G0,3

Outer Schedule

CPUs	Shuff/Deshuff	DMA	GPU
In0			
In1	Shf(In0 -> D0)		
In2	Shf(In1 -> D1)	DMA(In0->D0)	
In3	Shf(In2 -> D2)	DMA(In1->D1)	D0
In4	Shf(In3 -> D3)	DMA(In2->D2)	D1 E0
In5	Shf(In4 -> D4)	DMA(In3->D3) DMA(In'0<-F0)	D2 E1
In6	Shf(In5 -> D5) Dsf(In'0 <- F0)	DMA(In4->D4) DMA(In'1<-F1)	D3 E2
In7 In'0	Shf(In6 -> D6) Dsf(In'1 <- F1)	DMA(In5->D5) DMA(In'2<-F2)	D4 E3

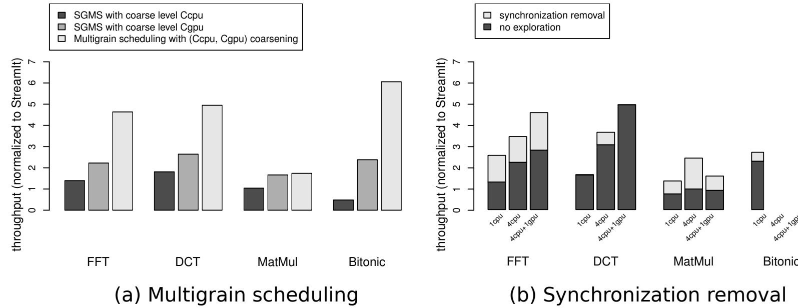
(b) Execution of the multigrain schedule.

Fig. 4. Multigrain Scheduling: : Each node is tagged with a couple (os, is), os represents the outer stage and is represent the inner stage. For each tick of the outer schedule, the inner schedule ticks $C_{gpu}/C_{cpu} = 4$ times. Yet the inner schedule works on chunks of data 4 times smaller, so the consumption-production rate is matched at the boundaries of the two schedules.

6 Experimental Results

We evaluated our proposed optimizations on four programs belonging to the StreamIT Benchmarks [1]: Fine-grained FFT, Direct Cosine Transform, Matrix Multiplication, and Bitonic Sort. Since we did not implement the StreamIt *peek* construct[15], we were limited to the benchmarks that did not use it.

The results presented in the following sections are throughputs normalized to a 1 CPU StreamIt baseline. The baselines were obtained by compiling the benchmarks with `strc --unroll 256 --destroyfieldarray --wbs` and running them on 1 CPU. The options passed to the StreamIt compiler are all the options in the `-O3` group, except `--partition` (removing this option in the Cluster backend has the effect of deactivating task fusion). We have deactivated task fusion in the StreamIt compilation since it has not yet been implemented in our compiler prototype. Task fusion eliminates data copies between nodes in the same partition so it has a significant impact on performance. Task fusion could be used in our method, by fusing the tasks inside each inner schedule; the evaluation of task fusion with the synchronization optimization and multigrain schedule will be the object of future works. Both the StreamIt baseline version and the version compiled by our prototype, were run on a sufficient number of iterations, to amortize start-up and pipeline filling costs. In our compiler chain the partitioning is delegated to the METIS graph partitioner [6].



(a) Multigrain scheduling (b) Synchronization removal

Fig. 5. (a) Evaluation of the multigrain scheduling on 4CPU+1GPU. All the measures were conducted on the same optimized graph. The first (respectively second) bar corresponds to a SGMS with the CPU (respectively GPU) optimal coarse level. The third bar is the multigrain scheduling which combines both optimal coarses.; (b) Evaluation of the bottleneck aware partitioning; the benchmark were scheduled with optimal multigrain scheduling.

6.1 Multigrain scheduling evaluation

We start by evaluating the gains of the multigrain schedule. For this we consider the communication optimized version of each benchmark graph. Each benchmark

is partitioned between the 4 CPU cores and the GPU as described in section 4. We then schedule a first version using a single SGMS working at the optimal CPU coarsening (C_{cpu}), we and a second version using a single SGMS working at the optimal GPU coarsening (C_{gpu}). Finally we schedule a third version using a multigrain schedule whose inner scheduling is coarsened with C_{cpu} and whose outer scheduling is coarsened with C_{gpu}.

As can be seen in figure 5(a), we obtain significant gains in FFT, DCT and Bitonic when using the Multigrain scheduling. This is expected: in the CPU optimized SGMS, the performance of the GPU partition is very degraded and becomes the bottleneck of the entire program. Similarly, in the GPU optimized SMGS, the CPU performance is degraded and slows down the GPU. The multigrain scheduling is able to adjust the working sizes of each partition and obtain an efficient schedule in an heterogeneous architecture.

In MatMul, the gains of the multigrain schedule are marginal. We attribute this to the fact that MatMul is a memory consuming benchmark. In the C_{gpu} profiling phase, the best C_{gpu} configuration found is 2048. Actually, higher values of C_{gpu} would probably give much better results, but they are discarded since they overflow the available memory in the GPU device. At the same time, the number of threads used in the GPU is small, since the number of registers used by the GPU partition is high. Therefore the GPU partition of MatMul is not very efficient, in fact it slows down the CPU only version, as will be seen in next section. The fact that C_{gpu} is low also means that the difference between C_{gpu} and C_{cpu} (64) is smaller than in the other benchmarks. Thus the gain of the multigrain scheduling is only marginal.

6.2 Synchronization optimization evaluation

We now evaluate the gains of the synchronization optimization presented in section 4. Each benchmark was compiled without and with synchronization optimizations for three different parallelism configurations: 1 CPU partition, 4 CPU partitions, 4 CPU + 1 GPU partitions. The programs were scheduled using multigrain scheduling with optimal coarse factors (*cf.* figure 5(b)).

Overall, the parallelization results are good for FFT, DCT and Bitonic; using all the cores, we obtain speed-up ranging from $\times 4.40$ to $\times 6.06$. MatMul obtains a $\times 2.42$ speedup for the 4 CPU version. However the GPU and CPU version only reaches a speedup of $\times 1.93$: as explained in the previous section, the GPU partition does not run in the optimal configuration since it would overflow the memory available in the GPU device. In fact it runs slower than one CPU partition, and becomes the bottleneck of the entire software pipeline.

The gains observed for the 1 CPU version by using the synchronization optimization, correspond to simplifications in the graph; since we are not tied by the series-parallel structure of StreamIt, we can replace some of the Split Join and Dup patterns with less costly alternatives. For example in 1, the top part of the original FFT butterfly graph (which changes reorder the input elements in an order appropriate for the FFT butterfly can be greatly simplified in the optimized version).

We attribute the gains observed for the 4 CPU and 4CPU + GPU versions to two effects. First there is the simplification of synchronization nodes which makes each partition more efficient and therefore also improves the parallelization throughput. There is also the fact that we reduce the communication costs between partitions. When the bottleneck is the DMA transfer between GPU and CPU, reducing the amount of communications between partitions may speed up the application. We can measure this effect for the Bitonic benchmark which is dominated by the communications. In the 1 CPU version, we gain a $\times 1.18$ factor after doing the synchronization removal, whereas in the 4 CPU+GPU version we gain a $\times 1.20$ factor.

The DCT benchmark is composed of two stages of filter nodes connected by a Split-Join junction. The synchronization removal is able to break this junction into smaller components. In fact in this case it does not reduce the amount of communications of the mapping, instead it breaks it down into many smaller transfers. In the original version we do one single big transfer, whereas in the transformed version we do many smaller ones. The transformed version improves slightly the 4 CPU version, we attribute this to a better behaviour of many small transfers regarding L3 cache. But when we add the GPU, the transformed version performs worse than the original program (3.52 normalized throughput). We attribute this to the way we implement DMA transfers: in our current prototype, each communication edge is mapped to a DMA transfer. In this case, doing one single big DMA transfer per schedule tick is preferable to many smaller DMA transfers. We believe this could be solved by fusing together all the DMA transfers in a Outer schedule tick.

7 Related Works

Streaming languages are based on the Synchronous Data Flow formalism [8] which provides a sound theoretical framework for ensuring the correct execution of data flow programs. StreamIt is both a language[15] and an optimizing compiler[4]. As in our approach synchronization optimization approach, StreamIt adapts the granularity and communications patterns of programs through graph transformations, which it separates in three classes: *(1)* Fusion transformations cluster adjacent filters, coarsening their granularity; *(2)* Fission transformations parallelize stateless filters decreasing their granularity; *(3)* Reordering transformations operate on splits and joins to facilitate Fission and Fusion transformations.

In this paper we use new reordering transformations that are more general than the ones in StreamIt since they are not constrained by the serie-parallel layout of the StreamIt graph. In this first version of our compiler, we have not considered fusing filters in the same partition, but we plan on implementing this feature, which should eliminate unnecessary data copies and increase the performance. Sermulins et al. show in[13] how to optimize StreamIt programs for cache effects in the context of a single core; in their paper the importance

of coarsening the program to tune for performance is studied and a cache aware fusing technique is proposed.

The authors of [7] were the first to apply Modulo Scheduling to stream graphs, evaluating their technique on the Cell BE. In [3] their work was extended considering an embedded target with memory and number of PE constraints. Carpenter et al. also work on executing Stream Graphs in an embedded context [2], they propose a graph partitioning mapping that preserves connectivity and convexity, to reduce the software pipeline depth and keep in the same partitions consumer and producers to enable efficient fusing transformations. The partitioning method they propose opportunely fuses or splits tasks, to load balance the work among the partitions.

Udupa et al.[16] successfully used Modulo Scheduling to compile stream graphs on GPU, and later on generalized their approach to hybrid GPU-CPU architectures. In [17] they propose an ILP based mapping that takes into account DMA costs and computation costs to find an optimum mapping in the architecture. Unlike us, they do not transform the graph structure before doing the partitioning. They also consider a single coarsening level for the entire graph, since they do not take into account the cache effects.

8 Conclusion

We propose in this paper a new method to optimize the throughput of stream programs on hybrid architectures. The contributions described are:

- A restructuration approach for stream graph, before partitioning and driven by an evaluation of synchronization costs. We have shown that this transformation improves the throughput of the graph by enabling better partitioning and reducing the amount of communication between partitions.
- A coarsening technique to tune parallelism grain independently for CPU and GPUs once the partitioning is achieved, in order to take into account the specific constraints on memory and DMA transfers for each architecture.

We also have developped an experimental compiler chain to validate our approach; the exploration of the graph variants is fast (less than 3 minutes on a desktop computer). The method proposed can be used with any partitioning method that considers the specificities of CPU/GPU architectures. We plan to investigate the interactions and benefits of our communication-aware approach combined to tasks optimizations such as fission/fusion techniques proposed in StreamIT. We also would like to determine if it is possible to derive the optimal coarse values C_{cpu} and C_{gpu} using a simple architectural model. This would remove the necessity of sampling profile runs.

References

1. Streamit benchmarks. `http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml`

2. Carpenter, P.M., Ramirez, A., Ayguade, E.: Mapping stream programs onto heterogeneous multiprocessor systems. In: CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems. pp. 57–66. ACM, New York, NY, USA (2009)
3. Choi, Y., Lin, Y., Chong, N., Mahlke, S., Mudge, T.: Stream compilation for real-time embedded multicore systems. In: Int. Symp. on Code Generation and Optimization. pp. 210–220. IEEE Computer Society, Washington, DC, USA (2009)
4. Gordon, M.I., Thies, W., Karczmarek, M., Lin, J., Meli, A.S., Lamb, A.A., Leger, C., Wong, J., Hoffmann, H., Maze, D., Amarasinghe, S.: A stream compiler for communication-exposed architectures. In: Int. Conf. on Architectural Support for Programming Languages and Operating Systems. pp. 291–303. ACM (2002)
5. Goubier, T., Blanc, F., Louise, S., Sirdey, R., David, V.: Définition du Langage de Programmation ΣC, RT CEA LIST DTSI/SARC/08-466/TG. Tech. rep. (2008)
6. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. 20(1), 359–392 (1998)
7. Kudlur, M., Mahlke, S.: Orchestrating the execution of stream programs on multicore platforms. In: Proc. of the SIGPLAN conf. on Programming Language Design and Implementation. pp. 114–124. ACM (2008)
8. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. 36(1), 24–35 (1987)
9. Liao, S.w., Du, Z., Wu, G., Lueh, G.Y.: Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In: Proc. of the Int. Symp. on Code Generation and Optimization (2006)
10. Lowerre, B.T.: The harpy speech recognition system. Ph.D. thesis, Pittsburgh, PA, USA (1976)
11. NVIDIA: NVIDIA CUDA Programming Guide 2.0 (2008)
12. de Oliveira Castro, P., Louise, S., Barthou, D.: Reducing Memory Requirements of Stream Programs by Graph Transformations. In: Intl. Conf. on High Performance Computing and Simulation (HPCS), (to appear). IEEE Computer Society (2010)
13. Sermulins, J., Thies, W., Rabbah, R., Amarasinghe, S.: Cache aware optimization of stream programs. In: LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems. pp. 115–126. ACM, New York, NY, USA (2005)
14. Thies, W.: Language and Compiler Support for Stream Programs. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (2009)
15. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction. pp. 179–196. Springer-Verlag, London, UK (2002)
16. Udupa, A., Govindarajan, R., Thazhuthaveetil, M.J.: Software pipelined execution of stream programs on gpus. In: Proc. of the 2009 Intl. Symp. on Code Generation and Optimization. pp. 200–209. IEEE Computer Society (2009)
17. Udupa, A., Govindarajan, R., Thazhuthaveetil, M.J.: Synergistic execution of stream programs on multicores with accelerators. In: Proc. of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems. pp. 99–108. ACM (2009)