Scheduling complex streaming applications on the Cell processor

Matthieu Gallet, Mathias Jacquelin, Loris Marchal

Scheduling complex streaming applications on the Cell processor

Matthieu Gallet [1,3,4] Mathias Jacquelin [1,3,4] Loris Marchal [2,4]

[1] ENS Lyon [2] CNRS [3] Université de Lyon

[4] LIP laboratory, UMR 5668, ENS Lyon – CNRS – INRIA – UCBL, Lyon, France

LIP Research Report RR-LIP-2009-29

Abstract

In this paper, we consider the problem of scheduling streaming applications described by complex task graphs on a heterogeneous multicore processor, the STI Cell BE processor. We first present a theoretical model of the Cell processor. Then, we use this model to express the problem of maximizing the throughput of a streaming application on this processor. Although the problem is proven NP-complete, we present an optimal solution based on mixed linear programming. This allows us to compute the optimal mapping for a number of applications, ranging from a real audio encoder to complex random task graphs. These mappings are then tested on two platforms embedding Cell processors, and compared to simple heuristic solutions. We show that we are able to achieve a good speed-up, whereas the heuristic solutions generally fail to deal with the strong memory and communication constraints.

1 Introduction

The last decade has seen the arrival of multicore processors in every computer and electronic device, from the personal computer to the high-performance computing cluster. Nowadays, heterogeneous multicore processor are emerging. Future processors are likely to embed several special-purpose cores –like networking or graphic cores– together with general cores, in order to tackle problems like heat dissipation, computing capacity or power consumption. Deploying an application on this kind of platform becomes a challenging task due to the increasing heterogeneity.

Heterogeneous computing platforms such as Grids have been available for a decade or two. However, the heterogeneity is now likely to exist at a much smaller scale, within a machine, or even a processor. Major actors of the CPU industry are already planning to include a GPU core

to their multicore processors [1]. Classical processors are also often provided with an accelerator (like GPUs, graphics processing units), or with processors dedicated to special computations (like ClearSpeed [7] or Mercury cards [18]), thus resulting in a heterogeneous platform. The best example is probably the IBM RoadRunner, the current leader of the Top500 ranking [21], which is composed of an heterogeneous collection of classical AMD Opteron processors and Cell processors.

The STI Cell BE processor is an example of such an heterogeneous architecture, since it embeds both a PowerPC processing unit, and up to eight simpler cores dedicated to vectorial computing. Moreover, the Cell processor is widely available and affordable. This is why we focus our study on this heterogeneous multicore processor.

Deploying an application on such a heterogeneous platform is not an easy task, especially when the application is not purely data-parallel. In this work, we focus on applications which exhibit some regularity, so that we can design efficient static scheduling solutions. We thus concentrate our work on streaming applications. These applications usually concern multimedia stream processing, like video edition softwares, web radios or Video On Demand applications [22, 14]. However, streaming applications also exist in other domains, like real time data encryption applications, or routing softwares, required by cell phones for example [20]. A stream is a sequence of data that have to go through several processing tasks. The application is generally structured as a directed acyclic task graph; this can be a simple chain of tasks, or a more complex structure, as illustrated in the following.

To process a streaming application on a heterogeneous platform, we have to decide which tasks will be processed on which processing elements, that is, to find a mapping of the tasks onto the platform. This is a complex problem since we have to take platform heterogeneity, task computing requirements, and communication volume into account. The objective is to optimize the *throughput* of the application: for example in the case of a video stream, we are looking for a solution that maximizes the number of images processed per time-unit.

Several stream solutions have already been developed or adapted for the Cell processor. DataCutter-Lite [13] is an adaptation of the DataCutter framework for the Cell processor, but it is limited to simple streaming applications described as linear chains, so it cannot deal with complex task graphs. StreamIt [12, 2] is a language developed to model streaming applications; a version of the Streamit compiler has been developed for the Cell processor, however it does not allow the user to specify the mapping of the application, and thus to precisely control the application. Some other frameworks allow to handle communications and are rather dedicated to matrix operations, like ALF [16], Sequoia [9], CellSs [6] or BlockLib [3].

In previous work, we have studied how to map complex workflows onto a set of heterogeneous resources, in the context of computing Grids. In particular, in [10], we explain how to compute an optimal mapping, that is a mapping reaching the optimal throughput. This is achieved using steady-state scheduling [5]: the activity of each resource (processing element or communication links) is bounded using linear constraints, and a mixed linear program (with integer and rational variables) gathering these constraints has to be solved to derive the optimal mapping. In the present paper, we show that steady-state scheduling meets the needs of an efficient scheduler for streaming application on heterogeneous processors, although new issues have to be solved due to

the completely different granularity of the tasks.

The rest of this paper is organized as follows: Section 2 presents in details the Cell processor together with a model of its operation, as well as our hypotheses for streaming applications. Then, we expose a theoretical complexity study of the problem in Section 3. Section 4 introduces some implementation choices that have an impact on the constraints for a good mapping. Section 5 then presents our optimal solution, obtained through the resolution of a mixed linear program. Finally, Section 6 presents the results of our experiments.

2 Platform and application model

In this section, we detail our view of the Cell processor, and introduce useful notations. In a second step, we also present the model of the streaming application.

2.1 Cell processor model

As said in the introduction, the Cell is a heterogeneous multicore processor. It has jointly been developed by Sony Computer Entertainment, Toshiba, and IBM [17], and embeds the following components:

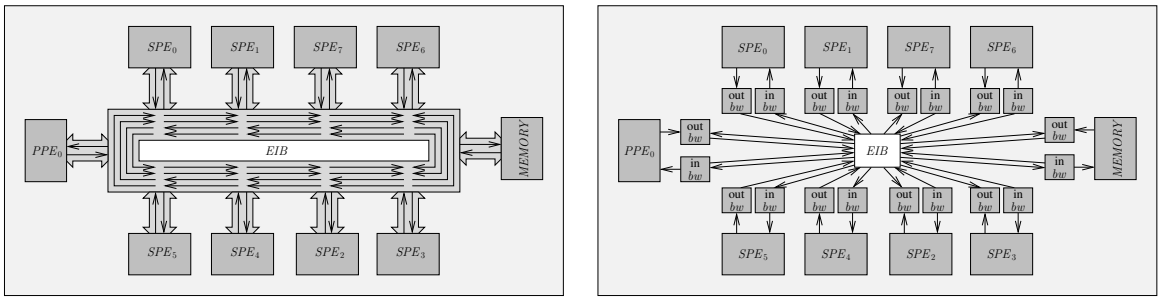
- **Power core.** Also known as the PPE (standing for Power Processing Element) and respecting the Power ISA 2.03 standard. This core is two-way multithreaded and its main role is to control the other cores, and to be used by the OS due to its similarity with existing Power processors. We do not *a priori* limit our study to only one Cell processor, so we consider platforms which may include n_P PPE cores $PPE_0, \ldots, PPE_{n_P}$.

- **Synergistic Processing Elements (SPE) cores.** These cores constitute the main innovation of the Cell processor and are small 128-bit RISC processors specialized in floating point, SIMD operations. These differences induce that some tasks are by far faster when processed on a SPE, while some other tasks can be slower. Each SPE has its own local memory (called *local store*) of size $LS = 256$ kB, and can access other local stores and main memory only through explicit asynchronous DMA calls. While the current Cell processor has eight SPEs, only six of them are available in the Sony PlayStation 3.Therefore, we consider any number n_S of SPEs in our model, denoted by $SPE_0, \ldots, SPE_{n_S}$.

- **Main memory.** Only PPEs have a transparent access to main memory. The dedicated memory controller is integrated in the Cell processor and allows a fast access to the requested data. Since this memory is by far larger than the SPE's local stores, we do not consider its limited size as a constraint for the mapping of the application.

- **Element Interconnect Bus (EIB).** This ring bus links all parts of the Cell processor to each other. The EIB has an aggregated bandwidth $BW = 200$ GB/s, and each component is connected to the EIB through a bidirectional interface, with a bandwidth $bw = 25$ GB/s in each direction.

All these components are displayed in a schematic view on Figure 1(a). To simplify formulas, we gather all processing elements under the same notation PE_i, so that the set of PPEs is $PPEs = \{PE_0, \ldots, PE_{n_P-1}\}$, while $SPEs = \{PE_{n_P}, \ldots, PE_{n_P+n_S-1}\}$ is the set of SPEs. Let n be the total number of processing elements, i.e., $n = n_P + n_S$. We have two classes of processing elements, which fall under to the *unrelated* computation model: a PPE can be fast for a given task T_k and slow for another one T_l, while a SPE can be slower for T_k but faster for T_l.



(a) Schematic view of the Cell processor.

(b) Theoretical view of the Cell processor.

Figure 1: Cell processor.

Communication model. Since the aggregated bandwidth of the EIB bus is equal to the sum of the bandwidth of all the interfaces (when we have all 8 interfaces), we assume in the following that this bus is not a constraint and that no contention occurs between communications. This is optimistic, since the structure of the bus may prevent irregular communication patterns to fully use its bandwidth, but we keep this model as a starting point. We will see in the evaluation that this model is precise enough to accurately compute an optimal schedule. As all communication elements are fully bidirectional, we use a bidirectional bounded-multiport model, with linear communication cost: a data of size S is sent (or received) in time S/b, where b is the bandwidth used for this communication, and the sum of incoming (respectively outgoing communications) of any element does not exceed its bandwidth in that direction. Note that memory accesses have to be counted as communications since they use the same interfaces.

Due to the limited size of the DMA stack on each SPE, each SPE can issue at most 16 simultaneous DMA calls. Similarly, each SPE has a separate stack for communications between itself and a PPE, which can handle at most eight simultaneous DMA calls.

Each core owns a dedicated communication interface (a DMA engine for the SPEs and a memory controller for the PPEs), and communications can thus be overlapped with computations.

Summarized theoretical model for the complexity study. In the following theoretical study, the Cell processor is therefore modeled by a set of processing elements as described on Figure 1(b). Each processing elements is provided with two communications interfaces in charge of the incoming and outgoing communications, with limited bandwidth $B_q^{\text{in}} = B_q^{\text{out}} = bw$. As we have seen above, this is the only contention point for communications. Communications can be overlapped with computations. Moreover, computation costs are modeled using the unrelated-machine model.

2.2 Application model

In this work, we target streaming applications. A stream is a sequence of *instances*, such as images in the case of a video stream. Our objective is to maximize the throughput of the application, that is the number of instances processed per time-unit.

As a simple example of a streaming application, we can think of a stream which must go through a simple chain of tasks, as depicted on Figure 2(a): in this application all instances of the stream must go through two tasks, such as two filters to apply to all images of a video stream. Streaming applications may be more complex, as described on Figure 2(b), and we model the structure of the application with a Directed Acyclic Graph $G_A = (V_A, E_A)$. The set V_A of nodes corresponds to tasks $T_1, \ldots, T_K$. The set E_A of edges models the dependencies between tasks, and the associated data: the edge from T_k to T_l is denoted by $D_{k,l}$. A data $D_{k,l}$, of size $data_{k,l}$ (in bytes), models a dependency between two task T_k and T_l, so that the processing of the ith instance of task T_l requires the data corresponding to the ith instance of data $D_{l,k}$ produced by T_k. Moreover, it may well be the case that T_l also requires the results of a few instances following the ith instance. In other words, T_l may need information on the near future (i.e., the next instances) before actually processing an instance. For example, this happens in video encoding softwares, when the program only encodes the difference between two images. We denote by $peek_k$ the number of such instances, that is, we need instances $i, i+1, \ldots, i+peek_k$ of $D_{k,l}$ to process the ith instance of T_l. This number of following instances is important not only when constructing the actual schedule and synchronizing the processing elements, but also when computing the mapping, because of the limited size of local memories where temporary data are stored.

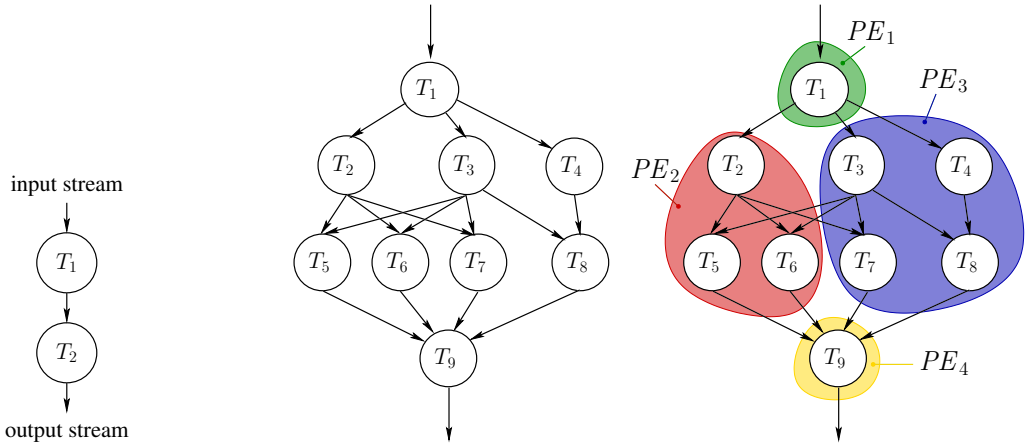
We also take into account communication between tasks and the main memory; we note $read_k$ the number of bytes read in memory by each instance of task T_k, and $write_k$ the number of bytes task T_k writes to memory.

Finally, since computing speeds of PPEs and SPEs are unrelated, $w_{PPE}(T_k)$ (respectively $w_{SPE}(T_k)$) denotes the time required for a PPE (resp. an SPE) to complete a single instance of T_k. As all SPEs and all PPEs are identical, these two values totally describe the computation requirements. Note that the processing times of tasks (as well as the size of the data exchanged by tasks) are instance independent.

3 Problem definition and complexity analysis

3.1 Definitions: mapping and schedule

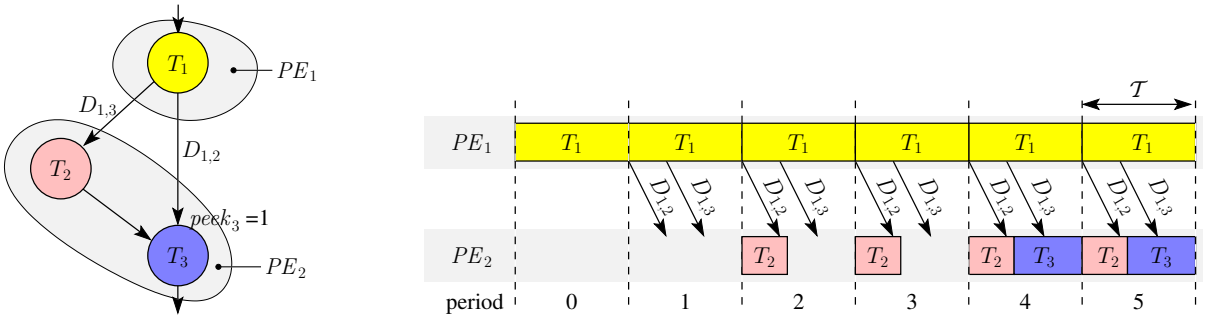
As explained above, our goal is to deploy a complex streaming application, described by a task graph, on a heterogeneous multicore architecture, under the objective of throughput maximization. To completely described the deployment of the application, we have to provide both a *mapping* and a *schedule*. The mapping describes *where* each task will be processed, whereas the schedule specifies *when*. Figure 2(c) shows one possible mapping for the task graph depicted on Figure 2(b) on four processing elements PE_1, PE_2, PE_3, and PE_4. Using this mapping, all instances of task T_1 are processed by PE_1, all instances of tasks T_2, T_5, and T_6 are processed by PE_2, etc. With



(a) Simple streaming application. (b) Other streaming application. (c) A possible mapping.

Figure 2: Applications and mapping.

such a mapping, all instances of a given task T_k are thus processed on the same processing element. We could also use a more general scheme, with different instances of the same task processed on different processing elements. This could help improving the throughput, but it would require to described the mapping of each instance of each task. We have studied this problem earlier, and provided a polynomial-time algorithm to solve it under some constraints on the task graph [4]. However, using a general mapping scheme has several drawbacks: it requires a complex flow control in order to route all messages, it induces larger buffers, and it involves a lot more communications when a task T_k with $peek_k > 0$ (i.e., a task which needs data from several instances to process a single instance) is replicated on several processing elements. Therefore, using a general mapping is not suited with the target architecture, which requires simple control and small communications, due to the limited size of the local memories. In this study, we thus concentrate on schedules describes by a simple mapping of tasks to processing elements.



(a) Application and mapping. $peek_1 = peek_2 = 0$ and $peek_3 = 1$

(b) Periodic schedule.

Figure 3: Mapping and schedule

Given a mapping, we can reconstruct a complete periodic schedule as illustrated on Figure 3.

After a few periods for initialization, each processing element enters a *steady state*: a processing element in charge of a task T_k simultaneously processes one instance of T_k, sends the result $D_{k,l}$ of the previous instance to the processing element in charge of each successor task T_l, and receives the data $D_{j,k}$ of the next instance from the processing element in charge of each predecessor task T_j. The exact construction of this periodic schedule is detailed in [4] for general mappings. In our case, the construction of the schedule is more straightforward: a processing element PE_i in charge of a task T_k simply processes it as soon as its input data is available, i.e., as soon as PE_i has received the data for the current instance and potentially the $peek_k$ following ones. Note that since we use a bounded-multiport model for communication, we do not need to precisely schedule the communications inside a period. All communications can happen simultaneously as soon as the average bandwidth needed during the period does not exceed the bandwidth bound on each interface. We denote by $\mathcal{T}$ the duration of a period in the obtained schedule. In this schedule, a new instance is processed every $\mathcal{T}$ time-units, thus the throughput of this schedule is $\rho = 1/\mathcal{T}$.

3.2 Complexity of the problem

In the previous section, we have seen that the mapping of an application onto the computing platform totally defines the schedule and its throughput ρ. In this section, we discuss the complexity of the problem of finding a mapping with optimal throughput. In this section, we forget about memory constraints, and constraints on the number of DMA transfers, as explained in Section 2.1: those constraints are not needed to prove that the problem is difficult. The associated decision problem is formally defined as follows.

Definition 1 (Cell-Mapping)**.** *Given a directed acyclic application graph G_A, a Cell processor with n_P PPE cores and n_S SPE cores and a bound B, is there a mapping with throughput $\rho \geq B$?*

Theorem 1. *Cell-Mapping is NP-complete in the strong sense.*

Proof. First note that we can check in polynomial time if a mapping has a given throughput: we simply have to make sure that the occupation time of each resource (processing element or communication interface) for processing one instance is not larger than $1/B$. Thus, this problem belongs to the NP class.

We prove the NP-completeness using a straightforward reduction from the Minimum Multiprocessor Scheduling problem restricted to two machines, which is known to be NP-complete even with a fixed number of machines [11]. An instance $\mathcal{I}_1$ of Minimum Multiprocessor Scheduling with two machines consists in n tasks $T_1, \ldots, T_n$ provided with a length $l(k,i)$ for each task T_k and each processor $i = 1, 2$, and a bound B'. The goal is to find an allocation f of the tasks on the processors such that $\sum_{k,f(k)=i} l(k,i) \leq B$ for each processor $i = 1, 2$.

We construct an instance $\mathcal{I}_2$ of our problem with one PPE (corresponding to processor 1) and one SPE (corresponding to processor 2). The streaming application consists in a simple chain of n tasks: $V_A = \{T_1, \ldots, T_n\}$ and there is a dependency $D_{k,k+1}$ for all $k = 1, \ldots n-1$. The computation times are $w_{PPE}(T_t) = l(t,1)$ and $w_{SPE}(T_t) = l(t,2)$, and communication costs are neglected: $data_{k,k+1} = 0$ for all $k = 1, \ldots n-1$. Finally, we set $B = 1/B'$. The construction of I_2 is obviously polynomial.

Assume that $\mathcal{I}_1$ has a solution f. Then, f defines a mapping of the tasks onto the processing elements. Using this mapping, the maximum occupation time of each processing element is at most B. Therefore, the throughput of this mapping is at least $1/B = B'$. f is thus a solution for instance $\mathcal{I}_1$. Similarly, a solution for $\mathcal{I}_2$ is to a solution for $\mathcal{I}_1$. □

The proof does not make use of communications:the problem is NP-complete even without any communication costs. Note that the restriction of Minimum Multiprocessor Scheduling with a fixed number of machines admits a fully polynomial approximation scheme (FPTAS) [15]. However, when considering a general application, communications have to be mapped together with computations, and the previous FPTAS cannot be applied. In spite of the NP-completeness of the problem, and thanks to the limited number of processing elements, we will show in Section 5 that a solution using Mixed Integer Programing allows to compute the optimal solution in a reasonable time.

4 Implementation choices

In this section, we relate the implementation choices made when designing our scheduler that impacts the optimization problem: we need to clarify these choices before we can express the constraints on memory, and the constraints on the number of communications from and to SPEs. There are two main issues: how to handle asynchronous communications and how to compute a bound on the buffer sizes.

4.1 Communications and DMA calls

The Cell processor has very specific constraints, especially on communications between cores. Even if SPEs are able to receive and send data while they are doing some computation, they are not multi-threaded and the computation must be interrupted to initiate a communication (but the computation is resumed immediately after the initialization of the communication). There are two ways to transfer data from a core to another:

1. The sender writes data into the destination local store;
2. The receiver reads data from the source local store. This method is a bit faster, so it is preferred to the former.

Due to the absence of auto-interruption mechanism, the thread running on each SPE has regularly to suspend its computation and check the status of current DMA calls. Moreover, as presented in Section 2.1, the DMA stack on each SPE has a limited size. A SPE can issue at most 16 simultaneous DMA calls, and can handle at most 8 simultaneous DMA calls issued by the PPEs. Furthermore, when building a steady-state schedule, we do not want to precisely order communications among processing elements, since it would require a lot of synchronizations. On the contrary, we assume that all the communications of a given period may happen simultaneously. These communications correspond to edges $D_{k,l}$ of the task graph when tasks T_k and T_l are not mapped on the same processing element. With the previous limitation on concurrent DMA calls,

this induces a strong limitation on the mapping: each SPE is able to receive at most 16 different data, and to send at most 8 data to PPEs per period.

4.2 Determining buffer sizes

Since SPEs have only 256 kB of local store, memory constraints on the mapping are tight. We need to precisely model them by computing the exact buffer sizes required by the application.

Mainly for technical reasons, the code of the whole application is replicated in the local stores of SPEs (of limited size LS) and in the memory shared by PPEs. We denote by $code$ the size of the code which is deployed on each SPE, so that the available memory for buffers is $LS - code$. A SPE processing a task T_k has to devote a part of its memory to the buffers for incoming data $D_{j,k}$, as well as for outgoing data $D_{k,l}$. Note that both buffers have to be allocated into the SPE's memory even if one of the neighbor tasks T_j or T_l is mapped on the same SPE. In a future optimization, we could save memory by avoiding the duplication of buffers for neighbor tasks mapped on the same SPE.

As we have seen in Section 3.1, before computing an instance of a task T_k, a processing element has to receive all the corresponding data, that is the data $D_{j,k}$ produced by each predecessor task T_j, both for the current instance and for the $peek_k$ following instances. Thus, the results of several instances need to be stored during the execution, because processing elements are not synchronized on the same instance. In order to compute the number of stored data, we first compute the index of the period in the schedule when the first instance of T_k is processed. The index of this period is denoted by $\mathit{firstPeriod}(T_k)$, and is expressed by:

$$\mathit{firstPeriod}(T_k) = \begin{cases} 0 & \text{if } T_k \text{ has no predecessor,} \\ \max\limits_{D_{j,k}} \left(\mathit{firstPeriod}(T_j)\right) + peek_k + 2 & \text{otherwise.} \end{cases}$$

All predecessors of an instance of task T_k are processed after $\max_{D_{j,k}} \left(\mathit{firstPeriod}(T_j)\right) + 1$ periods. We have also to wait for $peek_k$ additional periods if some following instances are needed, plus one period for the communication from the processing element handling the data, hence the result. By induction on the structure of the task graph, this allows to compute $\mathit{firstPeriod}$ for all tasks. For example, with the task graph and mapping described on Figure 3, we have $\mathit{firstPeriod}(1) = 0$, $\mathit{firstPeriod}(2) = 2$, and $\mathit{firstPeriod}(3) = 4$. Again, we could have avoided the additional period dedicated for communication when tasks are mapped on the same processor (e.g., we could have $\mathit{firstPeriod}(3) = 3$), but we let this optimization for future work to keep our scheduling framework simple.

Once the $\mathit{firstPeriod}(T_k)$ value of a task T_k is known, buffer sizes can be computed. For a given data $D_{k,l}$, the number of temporary instances of this data that have to be stored in the system if $\mathit{firstPeriod}(T_l) - \mathit{firstPeriod}(T_k)$. Thus, the size of the buffer needed to store this data is $buff_{k,l} = data_{k,l} \times (\mathit{firstPeriod}(T_l) - \mathit{firstPeriod}(T_k))$.

5 Optimal mapping through mixed linear programming

In this section, we present a mixed linear programming approach that allows to compute a mapping with optimal throughput. This study is adapted from [10], but takes into account the specific constraints of the Cell processor. The problem is expressed as a linear program where integer and rational variables coexist. Although the problem remains NP-complete, in practice, some softwares are able to solve such linear programs [8]. Indeed, thanks to the limited number of processing elements in the Cell processor, we are able to compute the optimal solution for task graphs of reasonable size (up to a few hundreds of tasks).

Our linear programming formulation makes use of both integer and rational variables. The integer variables are described below. They can only take values 0 or 1.

- α's variables which characterize where each task is processed: $\alpha_i^k = 1$ if and only if task T_k is mapped on processing element PE_i.
- β's variables which characterize the mapping of data transfers: $\beta_{i,j}^{k,l} = 1$ if and only if data $D_{k,l}$ is transfered from PE_i to PE_j (note that the same processing element may well handle both task if $i = j$).

Obviously, these variables are related. In particular, $\beta_{i,j}^{k,l} = \alpha_i^k \times \alpha_j^l$, but this redundancy allows us to express the problem as a set of linear constraints. The objective of the linear program is to minimize the duration $\mathcal{T}$ of a period, which corresponds to maximizing the throughput $\rho = 1/\mathcal{T}$. The intuition behind the linear program is detailed below. Remember that processing elements PE_0,...,PE_{n_P-1} are PPEs whereasPE_{n_P},...,PE_n are SPEs.

- Constraints (1a) define the domain of each variable: α and β lie in $\{0, 1\}$, while $\mathcal{T}$ is rational.
- Constraint (1b) states that each task is mapped on exactly one processing element.
- Constraint (1c) asserts that the processing element computing a task holds all necessary input data.
- Constraint (1d) asserts that a processing element can send the output data of a task only if it processes the corresponding task.
- Constraint (1e) ensures that the computing time of each PPE is no larger that $\mathcal{T}$, and Constraint (1f) does the same for each SPE.
- Constraint (1g) states that all incoming communication have to be completed within time $\mathcal{T}$, and Constraint (1h) does the same for outgoing communications.
- Constraint (1i) ensures that all temporary buffers allocated on the SPEs fit into their local stores.
- Constraint (1j) states that a SPE can perform at most 16 simultaneous incoming DMA calls, and Constraint (1k) makes sure that at most eight simultaneous DMA calls are issued by PPEs on each SPE.

$$
\left\{
\begin{array}{lll}
\multicolumn{3}{l}{\text{MINIMIZE } \mathcal{T} \text{ UNDER THE CONSTRAINTS}} \\
\text{(1a)} & \forall D_{k,l}, \forall PE_i \text{ and } PE_j, & \alpha_i^k \in \{0,1\}, \beta_{i,j}^{k,l} \in \{0,1\} \\
\text{(1b)} & \forall T_k, & \sum_{i=0}^{n-1} \alpha_i^k = 1 \\
\text{(1c)} & \forall D_{k,l}, \forall j, 0 \leq j \leq n-1, & \sum_{i=0}^{n-1} (\beta_{i,j}^{k,l}) \geq \alpha_j^l \\
\text{(1d)} & \forall D_{k,l}, \forall i, 0 \leq i \leq n-1, & \sum_{j=0}^{n-1} (\beta_{i,j}^{k,l}) \leq \alpha_i^k \\
\text{(1e)} & \forall i, 0 \leq i < n_P, & \sum_{T_k} (\alpha_i^k w_{PPE}(T_k)) \leq \mathcal{T} \\
\text{(1f)} & \forall i, n_P \leq i < n, & \sum_{T_k} (\alpha_i^k w_{SPE}(T_k)) \leq \mathcal{T} \\
\text{(1g)} & \forall i, 0 \leq i < n, & \alpha_i^k read_k + \sum_{D_{k,l}} \sum_{0 \leq j < n, j \neq i} (\beta_{j,i}^{k,l} data_{k,l}) \leq \mathcal{T} \times bw \\
\text{(1h)} & \forall i, 0 \leq i < n, & \alpha_i^k write_k + \sum_{D_{k,l}} \sum_{0 \leq j < n, j \neq i} (\beta_{i,j}^{k,l} data_{k,l}) \leq \mathcal{T} \times bw \\
\text{(1i)} & \forall i, n_P \leq i < n, & \sum_{T_k} \left(\alpha_i^k \left(\sum_{D_{k,l}} buff_{k,l} + \sum_{D_{l,k}} buff_{l,k} \right) \right) \leq LS - code \\
\text{(1j)} & \forall j, n_P \leq j < n, & \sum_{0 \leq i < n, i \neq j} \sum_{D_{k,l}} \beta_{i,j}^{k,l} \leq 16 \\
\text{(1k)} & \forall i, n_P \leq i < n, & \sum_{0 \leq j < n_P} \sum_{D_{k,l}} \beta_{i,j}^{k,l} \leq 8
\end{array}
\right. \tag{1}
$$

We denote $\rho_{opt} = 1/\mathcal{T}_{opt}$, where $\mathcal{T}_{opt}$ is the value of $\mathcal{T}$ in any optimal solution of Linear Program (1), with $O(n^2)$ variables and $O(n^2)$ constraints. The following theorem states that ρ_{opt} is the maximum achievable throughput.

Theorem 2. *An optimal solution of Linear Program* (1) *describes a mapping with maximal throughput.*

Proof. Consider an optimal solution of Linear Program (1), with throughput $\rho_{opt} = 1/\mathcal{T}_{opt}$. Thanks to the constraints of the linear program, this solution defines a valid mapping with period $\mathcal{T}_{opt}$. As we have seen before, this corresponds to a schedule with throughput ρ_{opt}.

We now consider any possible mapping of the application onto the processing elements, and the associated schedule. We define $\alpha_i^k = 1$ if task T_k is mapped on processing element PE_i in this mapping, and 0 otherwise. We also define $\beta_{i,j}^{k,l} = \alpha_i^k \times \alpha_j^l$. We finally consider the period $\mathcal{T}$ achieved by the schedule associated with this mapping. Since $\mathcal{T}$ is a valid period for this mapping, these variables satisfies all constraints of Linear Program (1). α, β and $\mathcal{T}$ are then a solution of the linear program. Thus, $\mathcal{T} \leq \mathcal{T}_{opt}$ and this mapping has a throughput $1/\mathcal{T}$ which is not larger than the ρ_{opt}. □

6 Experimental validation

To assess the quality of both our model and our scheduling strategy, we conduct several experiments. We used two hardware platforms: a Sony PlayStation 3, and an IBM QS22. The Sony PlayStation 3 is a video game console built around a single Cell processor, with only 6 usable SPEs and a single Power core. The IBM QS22 is a high performance computing server built on

top of two Cell processors connected through a high performance interface and sharing main memory. Using both Cell processors generates many new difficulties, both for software development (moving threads on particular PPEs) and platform modeling (especially for communication contention between both Cell processors). Therefore, we first focus on optimizing the performance for a single Cell processor, and we will adapt our study to several processors in a future work. Thus, in the experiments, the number of PPE is $n_P = 1$ and the number of SPEs n_S may vary from zero to eight (or six on the PlayStation 3). For our experiments, we use ILOG CPLEX [8] to solve the linear program introduced in the previous section. In order to reduce the computation time for solving the linear program, we used the ability of CPLEX to stop its computation as soon as its solution is within 5% of the optimal solution. While this significantly reduces the average resolution time, it still offers a very good solution. Using this feature, the time for solving a linear program was always kept below one minute (mostly around 20 seconds), which is negligible in front of the duration of a stream application.

6.1 Scheduling software

Together with these hardware platforms, we also need a software framework to execute our schedules while handling communications. If there already exist some frameworks dedicated to streaming applications [12, 13], none of them is able to deal with complex task graphs while allowing to statically select the mapping. Thus, we have decided to develop one[1]. Our scheduler only requires as input parameters the description of the task graph, its mapping on the platform, and the code of each task. Even if it was designed to use the mapping returned by the linear program, it can also use any other mapping, such as the heuristic strategies described below.

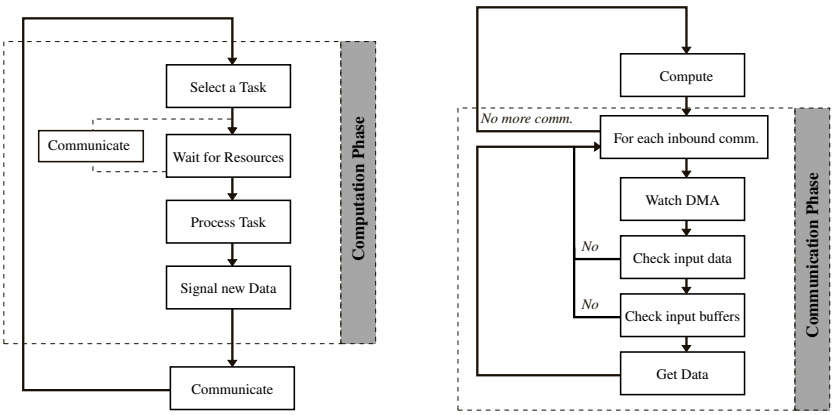
We now briefly describe our scheduler, which is mainly divided into two main phases: the *computation phase*, during which the scheduler selects a task and processes it, and the *communication phase*, during which the scheduler performs asynchronous communications. These steps, depicted on Figure 4, are executed by every processing element. Moreover, since communications have to be overlapped with computations, our scheduler cyclically alternates between those two phases.

The computation phase, which is shown on Figure 4(a), begins with the selection of a runnable task according to the provided schedule, then it waits for the required resources (input data and output buffers) to be available. If all required resources are available, the selected task is processed, otherwise, it moves to the communication phase. Whenever new data is produced, the scheduler signals it to every dependent processing elements.

The communication phase, depicted in Figure 4(b), aims at performing every incoming communication, most often by issuing DMA calls. Therefore, the scheduler begins by watching every previously issued DMA call in order to unlock the output buffer of the sender when data had been received. Then, the scheduler checks whether there is new incoming data. In that case, and if enough input buffers are available, it issues the proper "Get" command.

To obtain a valid and efficient implementation of this scheduler, we had to overcome several issues due to the very particular nature of the Cell processor. First, the main issue is heterogeneity:

[1] An experimental version of our scheduling framework is available online, at `http://graal.ens-lyon.fr/~mjacquel/cell_ss.html`



(a) Computation Phase. (b) Communication Phase.

Figure 4: Scheduler state machine.

the Cell processor is made of two different types of cores, which induces additional challenges for the programmer:

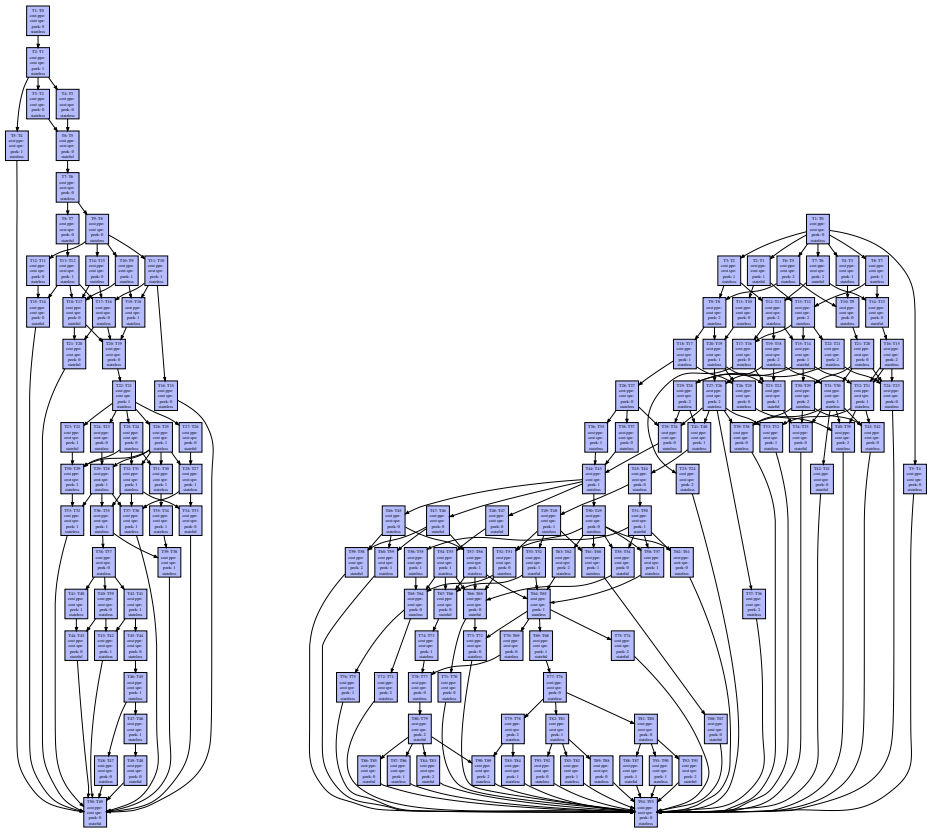
- SPE are 32-bit processors whereas the PPE is a 64-bit architecture;
- Different communication mechanisms have to be used depending on which types of processing elements are implied in the communication. To properly issue our "Get" operations, we made use of three different intrinsics: `mfc_get` for SPE to SPE communications, `spe_mfcio_put` for SPE to PPE communication, and `memcpy` for communication between PPE and main memory.

Another difficulty lies in the large number of variables that we need to statically initialize in each local store before starting the processing of the stream: the information on the mapping, the buffer for data transfer, and some control variables such as addresses of all memory blocks used for communications. This initialization phase is again complicated by the different data sizes between 32-bit and 64-bit architectures, and the runtime memory allocation.

All these issues show that the Cell processor is not designed for such a complex and decentralized usage. However, our success in designing a complex scheduling framework proves that it is possible to use such a heterogeneous processor for something else than pure data-parallelism.

6.2 Applications

We test our scheduling framework on three random task graphs, obtained with the DagGen generator [19]. This allows us to test our strategy against task graphs with different depths, widths, and branching factors. Two of the three generated task graphs are described on Figures 5(a) and 5(b), and the last one is a simple chain graph with 50 tasks. For these graphs, we generated 6 variants of different communication-to-computation ratio (CCR), resulting in 24 different random applications. We compute the CCR of a scenario as the total number of transfered elements divided by the number of operations on these elements. In the experiments, the CCR goes from 0.775 (computation-intensive scenario) to 4.6 (communication-intensive scenario).



(a) Random graph 1

(b) Random graph 2

Figure 5: Two random task graphs used in the experiments

6.3 Reference heuristics

Here, we present two simple heuristic strategies that we have designed in order to assess the performance of our schedule based on linear programming. These strategies are designed for the particular hardware platforms used in our tests; therefore, they can handle a Cell processor with several SPEs and one PPE. Since we noticed that memory limitation of the SPEs is one of the most significant factor for performance, these strategies focus on a reasonable usage of this limited memory. The second one also takes the computation amount into account when mapping tasks to processing elements. Both strategies are *greedy* strategies: they map the tasks one after the other, and never go back on a previous decision.

The first heuristic, called GREEDYMEM, process the tasks in a topological order. Given a task, it select the SPEs which have enough free memory to host the tasks and its buffers. Among those SPEs, the one with the least loaded memory is chosen. If no SPE can host the task, it is allocated on the PPE.

The second heuristic is called GREEDYCPU and is very similar to GREEDYMEM: among the processing elements (SPEs and PPE) with enough memory to host a task, it selects the one with the smallest computation load.

6.4 Experimental results

The results presented here are obtained on the QS22, using up to eight SPEs. We have also performed the same experiments on the PlayStation 3, and the results are exactly the same as the one on the QS22 (using six SPEs). Thus, we show only the results on the QS22.

6.4.1 Entering steady-state

First, we show that our scheduling framework succeeds in reaching steady-state, and that the throughput is then similar to the one predicted by the linear program. Figure 6 shows the experiments done with the task graph described in Figure 5(a), with a CCR of 0.775, on the QS22 using all eight SPEs. We notice that a steady-state operation is obtained after approximately 1000 instances. Note that one instance consists only of a few bytes, so the steady-state throughput is obtained quickly compared to the total length of the stream. In steady state, the experimental throughput achieves 95% of the throughput predicted by the linear program. The small gap is explained by the overhead of our framework, and the synchronizations induced when communications are performed.

6.4.2 Comparing heuristics with linear program

Then, we compare the throughput of the mapping obtained through mixed linear programming with the mappings computed by the heuristics. We compute the speed-up obtained by each mapping, that is the achieved throughput normalized to the throughput when using only the PPE. Figure 7 shows the results for the three random task graphs, with CCR 0.775, for different numbers of SPEs used.

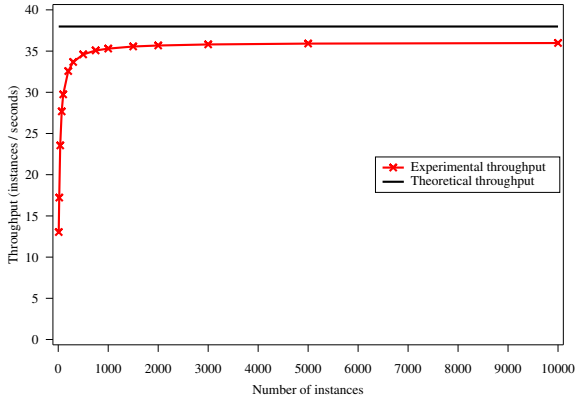
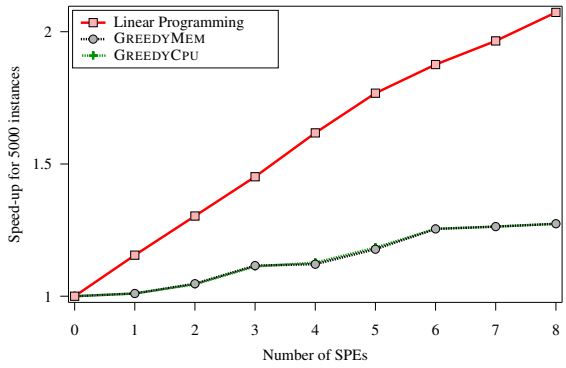
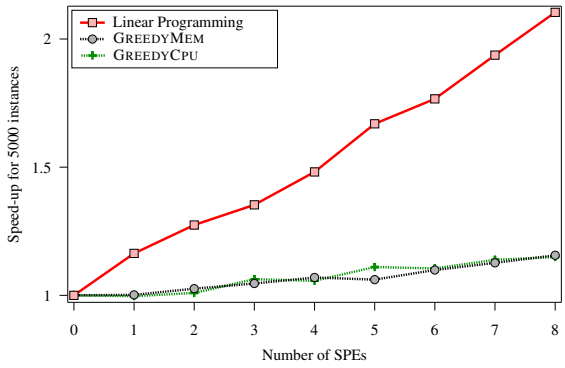


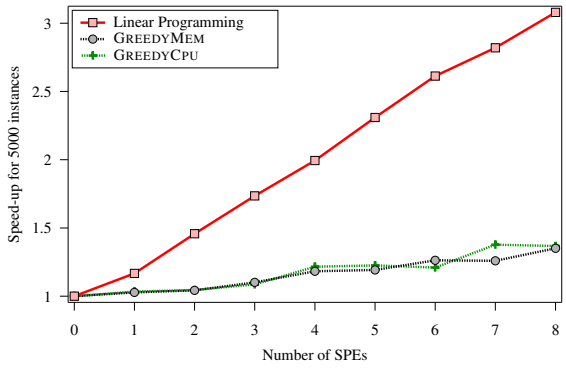
Figure 6: Throughput achieved depending on the number of instances.



(a) Speed-up for random graph 1



(b) Speed-up for random graph 2



(c) Speed-up for random graph 3

Figure 7: Speed-up obtained on the QS22 depending on the number of SPEs used.

Mappings generated through linear programming achieves the best speed-up, and they also offer the best scalability among every mappings. For these task graphs, we reach a speed-up between 2 and 3 using 8 SPEs, while the heuristics reaches a maximum speed-up of 1.3. This shows that it is crucial to take data transfers into account when designing mapping strategies. However, our complex strategy based on linear programming and using a better model of the platform is able to get good performance out of this complex architecture.

6.4.3 Influence of the communication-to-computation ratio

We now test the performance of the mapping computed with the mixed linear program, for different values of the communication-to-computation (CCR). The speed-up when using the 8 SPEs of the QS22 are presented on Figure 8. We see that the larger the CCR, the more difficult to get a good speed-up. Indeed, when communications are predominant, it is hard to distribute tasks among SPEs and to reach a decent throughput. Eventually, the best policy is to map all tasks to the PPE.

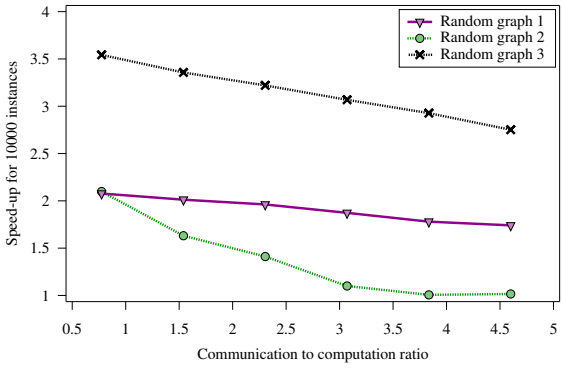


Figure 8: Speed-up of random task graphs according to the CCR, for the mapping computed through linear programming using 8 SPEs on an QS22.

In conclusion, our experiments demonstrate the usefulness of our approach on the Cell. High speed-ups are obtained whenever the CCR of the application is sufficiently low, and our mappings offer better performance and scalability compare to the heuristics we have implemented.

7 Conclusion

In this paper, we have studied the scheduling of streaming applications on a heterogeneous multicore processor: the STI Cell processor. The first challenge was to come up with a realistic and yet tractable model of the Cell processor. We have designed such a model, and we have used it to express the optimization problem of finding a mapping with maximal throughput. This problem has been proven NP-complete, and we have designed a formulation of the problem as a mixed linear program. By solving this linear program with appropriate tools, we can compute a mapping with optimal throughput.

In a second step, we have implemented a complete scheduling framework to deploy streaming applications on the Cell processor. This framework allows the user to deploy any streaming application, described by a potentially complex task graph, on a Cell processor, given any mapping of the application to the platform. Thanks to this scheduling framework, we have been able to test our scheduling strategy, and to compare it to simpler heuristic strategies. We have shown that our strategy reaches 95% of the throughput predicted by the linear program, that our approach has a good and scalable speed-up when using up to 8 SPEs, and that our strategy clearly outperforms the simple heuristics, which are unable to deal with the complex mapping problem. Overall, this demonstrates that scheduling a complex application on a heterogeneous multicore processor is a challenging task, but that scheduling tools can help to achieve good performance.

This work has several natural extensions, and we have already started to study some of them. First, several optimizations of the scheduling framework could be implemented to achieve even better performance, such as limiting buffer sizes for neighbors tasks mapped on the same processing elements. Then, we need to extend and refine the model we have presented in this paper to more complex platforms. For example, we would like to be able to use both Cell processors of the QS22, or even a cluster of QS22 machines. On the long view, we would like to adapt our model and framework to other heterogeneous multicore platforms. Finally, we have seen that simple heuristics fail to efficiently map the application on the platforms; thus it would be interesting to design involved mapping heuristics which approach the optimal throughput.

References

[1] AMD Fusion. `http://fusion.amd.com`.

[2] Streamit project. `http://groups.csail.mit.edu/cag/streamit/index.shtml`.

[3] M. Ålind, M. Eriksson, and C. Kessler. BlockLib: a skeleton library for Cell broadband engine. In *IWMSE '08: 1st international workshop on Multicore software engineering*, pages 7–14, New York, NY, USA, 2008. ACM.

[4] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters. *International Journal of Foundations of Computer Science*, 16(2):163–194, 2005.

[5] Olivier Beaumont and Loris Marchal. Steady-state scheduling. In *Introduction to Scheduling*. Chapman and Hall/CRC Press, 2009. To appear.

[6] P. Bellens, J. Perez, R. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *SC'06: ACM/IEEE Super Computing Conference*, pages 5–5, Nov. 2006.

[7] ClearSpeed. ClearSpeed technology. `http://www.clearspeed.com/technology/index.php`.

[8] ILOG CPLEX: High-performance software for mathematical programming and optimization. `http://www.ilog.com/products/cplex/`.

[9] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Reiter Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. *SC'06: ACM/IEEE Super Computing Conference*, 0:4, 2006.

[10] Matthieu Gallet, Loris Marchal, and Frédéric Vivien. Efficient scheduling of task graph collections on heterogeneous resources. In *International Parallel and Distributed Processing Symposium IPDPS'2009*. IEEE Computer Society Press, 2009.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[12] X. Hang. A streaming computation framework for the Cell processor. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, Aug 2007.

[13] T. Hartley and U. Catalyurek. A component-based framework for the Cell broadband engine. In *IPDPS'09: International Parallel and Distributed Processing Symposium*, Los Alamitos, CA, USA, june 2009. IEEE Computer Society Press.

[14] Stephen L. Hary and Fusun Ozguner. Precedence-constrained task allocation onto point-to-point networks for pipelined execution. *IEEE Trans; Parallel and Distributed Systems*, 10(8):838–851, 1999.

[15] Ellis Horowitz and Sartaj Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, 23(2):317–327, 1976.

[16] IBM. Accelerated library framework. `http://www.ibm.com/developerworks/blogs/page/powerarchitecture?entry=ibomb_alf_sdk30_1&S_TACT=105AGX16&S_CMP=EDU`, 2007.

[17] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David J. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, 2005.

[18] Mercury. Mercury technology. `http://www.mc.com/technologies/technology.aspx`.

[19] F. Suter. Dag generation program. `http://www.loria.fr/\~suter/dags.html`, 2009.

[20] William Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2001.

[21] TOP500 Project. June 06 list. `http://www.top500.org/lists/2009/06`.

[22] Q. Wu, J. Gao, M. Zhu, N.S.V. Rao, J. Huang, and S.S. Iyengar. On optimal resource utilization for distributed remote visualization. *IEEE Trans. Computers*, 57(1):55–68, 2008.