ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK
Institut für
Technische Informatik und
Kommunikationsnetze

Semester Thesis
at the Department of Information Technology
and Electrical Engineering

Development of a Streaming Application Benchmark Set

Georgia Giannopoulou

Advisors: Wolfgang Haid, Kai Huang
Professor: Prof. Dr. Lothar Thiele

Zurich, 17th February 2010

Abstract

Future requirements of embedded systems can only be fulfilled using multiprocessor systems. One challenge of these systems is to efficiently program applications. The DOL framework addresses this problem by allowing the specification of applications as Kahn process networks and by mapping these applications to multiprocessor architectures.

This semester thesis proposes a set of benchmark applications which can be used to evaluate the peak computational performance and peak communication bandwidth of applications executing on several heterogeneous multiprocessor systems in which inter-processor communication is achieved via message passing.

These applications have been designed as synchronous data flows and have been developed within the DOL framework. In a second step, several alternative configurations of them have been used to evaluate the performance of the Sony/Toshiba/IBM Cell Broadband Engine. To this end, the existing DOL code generation back-end for the CBE has been revisited and improved so that the corresponding run-time environment can be used as a reliable platform for experiments.

Acknowledgements

I would like to express my sincere gratitude to Prof. Dr. Lothar Thiele for giving me the opportunity to write this semester thesis at the Computer Engineering Laboratory.

I would also like to thank my advisors Wolfgang Haid and Kai Huang for their continuous support and valuable discussions during progression of this work. Their suggestions and constructive feedback guided me throughout this thesis. It was a pleasure collaborating with them.

Contents

List of Figures

Listings

1 Introduction

The advancements in semiconductor technologies have led to the emergence of the system-on-chip (SoC), which enables more and more functionality to be integrated on a single chip. With the increasing complexity of modern applications (e.g. multimedia, telecommunications, signal processing), traditional single processor architectures can no longer meet the demanding performance requirements. Nowadays, the trend of embedded system design is moving from single processor architectures toward heterogeneous multiprocessor SoC architectures. This shift calls for new methodologies as traditional ad-hoc approaches fall short of dealing with the complexity and heterogeneity of multiprocessor SoCs. The design of future MPSoCs requires a scalable hardware-software design approach, including scalable applications, scalable architectures and scalable design techniques.

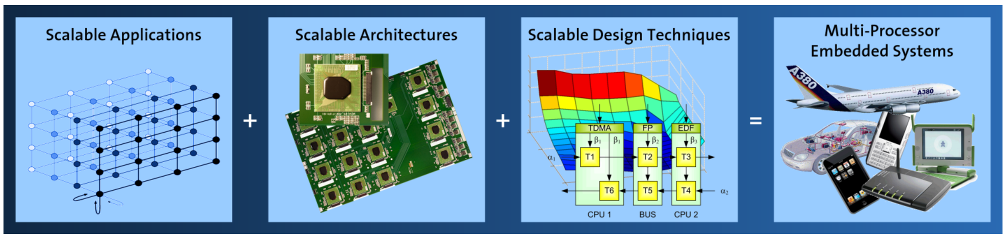


Figure 1.1.: Main aims of the SHAPES project

1.1. Motivation and Contribution

In the previously described context, there is always a need for applications that can be used to evaluate and demonstrate the consequences of newly developed techniques. Examples are:

- evaluate accuracy of formal performance analysis methods by comparing results from simulation and formal analysis for different applications
- evaluate performance of MPSoC run-time environments by running different applications
- evaluate performance of design space exploration by comparing predicted performance and actual performance for different applications and different configurations of the same application

In order to support these activities, the main contribution of this semester project has been the design and implementation of a benchmark set consisting of streaming applications that can be easily parametrized to efficiently execute on different MPSoCs. In particular, the applications are targeted at heterogeneous MPSoCs with a distributed memory architecture where the individual processors communicate via message passing. Execution of the applications has been demonstrated on a multiprocessor platform, the Sony/Toshiba/IBM Cell Broadband Engine, to evaluate their performance.

1.2. Background

1.2.1. SHAPES Project

The European SHAPES (Scalable Software/Hardware Architecture Platform for Embedded Systems) project [1] aims at developing a complete MPSoC hardware platform and the according design flow [2]. The main objectives of the SHAPES project are to investigate the tiled hardware paradigm, to experiment with real-time and communication aware system software, and to validate the hardware and system platform through a set of benchmarking applications.

The SHAPES hardware is a new MPSoC architecture, which intends to be highly scalable in computation power and communication capacity. A tiled architecture consists of predefined processing tiles which are connected to each other. SHAPES uses heterogeneous tiles, which consist of a RISC processor, a VLIW (very long instruction word) DSP, a distributed network processor and on-tile memories and peripherals [1]. A very important issue is *scalability*, which means that an application should be portable to any different SHAPES hardware architecture without much effort. Specifically, it should be possible to map an application onto architectures with largely different amounts of tiles.

The software stack is composed of three layers: the application, the operating system (OS) and the hardware dependent software (HdS). The two lower layers, the OS and the HdS, run locally on each processor. Another part of the software framework is the distributed operation layer (DOL), the purpose of which is to map the application to the underlying multiprocessor architecture. DOL is discussed in more detail later in this chapter.

The tiled architecture approach has certain advantages, it is however difficult for an application to fully exploit its potential. There may be long delays between distant tiles, overloaded communication resources, or the application may not expose enough parallelism. In all these cases, the architecture's full computing power cannot be exploited. The system software therefore has to make sure that the applications are executed efficiently on the SHAPES hardware, while minimizing the effort required for the application programmer. This is the main software challenge. Two key points are considered: first, as the system itself is highly parallel, so should be the application. The application programmer must be able to fully expose the algorithm's parallelism to the SHAPES platform, which makes it necessary to break with the conventional way of writing an application. But even if the application is written in a way that exposes the parallelism well, this information about the algorithmic structure must be preserved by the SHAPES system software. Second, the system software must be fully aware of important architectural parameters like bandwidth, computing capabilities and latencies [3], [4], [5].

1.2.2. Distributed Operation Layer

As has already been mentioned, one major software aspect regarding MPSoC design is finding an optimal mapping for an application onto an allocated hardware architecture. SHAPES' distributed operation layer (DOL) [6] is a platform-independent programming framework, developed at the Computer Engineering Laboratory at Swiss Federal Institute of Technology (ETH), which addresses this challenge.

To use DOL to find a mapping, the designer must specify the application to be mapped. The specification exposes the parallelism of the application and is completely separated from any architectural aspects. DOL uses Kahn process networks [7] as the model of computation. A process network is composed of processes which are connected by first-in first-out (FIFO) channels. Processes can only perform local computations, read data from input channels and write data to output channels. The structure of the process network, a directed graph whose nodes represent processes and whose directed edges represent communication channels, has to be specified in XML. The functionality of the application is defined by the behaviour of the processes. Each process has to be specified in plain C/C++ whereby a set of DOL specific coding rules has to be respected.

A specification has to be given for the target architecture too. It is also defined in XML and contains structural, performance, and parametric data. The structure specifies the platform's resources such as processors, memories, hardware channels and their interconnections. Performance data give information about clock frequencies, communication delays, throughputs and therelike. Parametric data can for example define memory sizes or operating system parameters. As the mapping optimization is performed at the system level, such an abstract representation of the architecture is sufficient.

The application and architecture specifications are the input of DOL. To obtain a base for optimization decisions, profiling data of the application and the hardware are collected first. Functional simulation of the application reveals the number of process invocations and the amount of data transmitted over each channel in the specified process network.

An iterative design space exploration and estimation cycle then tries to find optimal mappings of the application onto the architecture. Mapping includes the binding of processes to processors and communication channels to hardware channels as well as scheduling of shared resources. Exploration of mappings is based on evolutionary algorithms. Candidate solutions are then analyzed and simulated. Performance estimation results are fed back into optimization for further improvements. Mappings are also represented in XML.

DOL targets an efficient execution of parallel applications on a heterogeneous MPSoC. It makes the design process scalable and keeps it flexible. The resulting mapping can be used by other tools to generate the program code for the different processors. An overview about the role of the DOL framework in the MPSoC design flow is given in Figure 1.2 [8], [9].

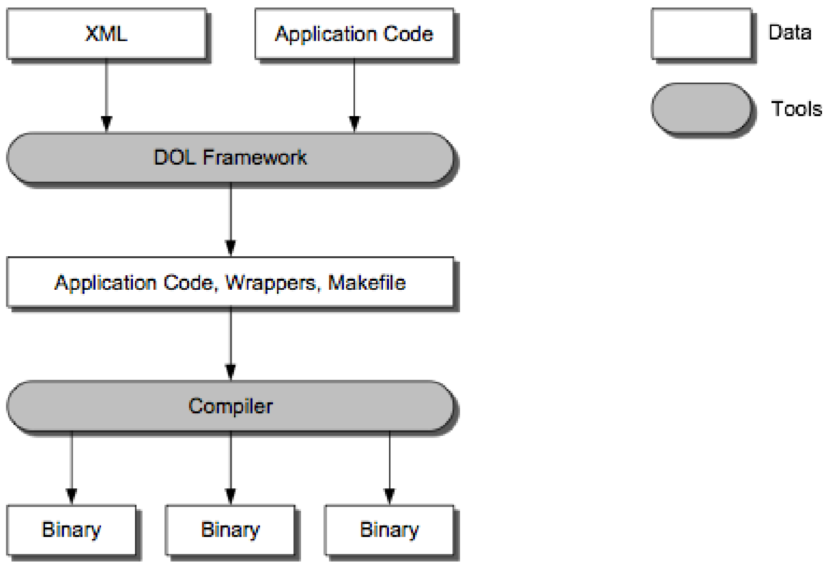


Figure 1.2.: Flowchart of the DOL Framework

1.3. Related Work

A large number of benchmark suites, consisting of general-purpose and/or streaming applications, have been developed for the evaluation of multicore systems. Only a few of them, though, have been designed to target heterogeneous MPSoCs with a distributed memory architecture, where inter-process communication is achieved via message passing. Below some well-known benchmarks are listed along with a short explanation of the reasons why they do not correspond to the specifications of the envisioned streaming benchmark suite.

- EEMBC MultiBench [10]: Designed to evaluate scalable symmetrical multicore processor (SMP) architectures with shared memory.
- PARSEC [11], [12]: Designed to evaluate shared-memory chip-multiprocessors (CMPs).

- MediaBench II [13]: Designed to evaluate shared-memory video processors and systems.
- ALPBench [14]: Incorporating complex multimedia applications to be run over shared-memory multiprocessor architectures.
- MPIBench [15], SKaMPI [16]: Targeting heterogeneous platforms with distributed memory, focusing on performance measurement of message passing interface (MPI) communication routines.
- PARKBENCH [17]: Implementing computationally intensive general-purpose (not streaming) applications to be run on heterogeneous distributed-memory multicore systems.
- STREAM Benchmark [18]: Single synthetic benchmark program targeting both shared- and distributed-memory parallel architectures, focusing on measurement of sustainable memory bandwidth of them.
- StreamIt [19]: Including benchmarks in StreamIt programming language for the evaluation of streaming optimizations (made by StreamIt compiler) and architectures. This benchmark suite seems to be the one closest to our goal, however translation of StreamIt code into C/C++ requires considerable effort.

1.4. Outline

The remaining part of this thesis is organised as follows. In Chapter 2, the benchmark applications which are proposed to measure the peak performance as well as the peak communication bandwidth of diverse MPSoC architectures are presented. In Chapter 3, these applications are used to evaluate the run-time environment of the DOL for the Cell Broadband Engine. Chapter 4 concludes the thesis by summarizing its main contributions and outlining possible future work.

2 Benchmark Applications

2.1. Application Requirements

This section describes the requirements of the envisioned benchmark set of streaming applications.

The applications belonging to this set should be implemented in a platform-independent manner as Kahn process networks in the DOL framework. This means that each application is expressed as a network of concurrently executing processes that communicate via point-to-point FIFO channels. While the distributed operation layer allows to specify general Kahn process networks, the benchmark applications should restrict to the synchronous dataflow model (SDF) [20]. The benchmark suite should contain applications that allow to determine the following properties of an MPSoC (and the run-time environment running on top of it):

1. peak performance in terms of floating point operations per second
2. peak bandwidth of FIFO communication between processors and peak aggregate bandwidth of FIFO communication on the entire MPSoC

The benchmark applications should be executable on the following architectures that differ with respect to the computation, communication, and memory resources:

- Sony/Toshiba/IBM Cell Broadband Engine [21]
- Atmel DIOPSIS 940 [3]
- MPARM [22]
- 32-bit and 64-bit AMD and Intel processors

Based on existing run-time environments and software synthesis back-ends contained in the distributed operation layer, the different platform-specific implementations can be automatically derived from the platform-independent application specification. This results in the following requirements for the specification of each application:

- Platform independence: Due to the different target architectures, the applications are written in a platform-independent manner. This regards 32-/64-bit compatibility as well as the endianness of data. For instance, the Cell Broadband Engine contains 64-bit processors with big endian byte order whereas the ARM9 processor contained in the Atmel DIOPSIS 940 is a 32-bit processor with little endian byte order.
- Scalability: The applications will be executed on different platforms with different architectural characteristics. Applications are easily parametrizable with respect to the problem size, degree of parallelism, and the granularity of processes and communication.
- Parallelism: The applications expose parallelism as much as possible to enable efficient parallel execution.
- Communication: Each application is implemented using standard FIFO channels and windowed FIFO channels for inter-process communication. (For a definition of windowed FIFO channels, see Section 3.1)

For each benchmark application, the following items need to be available:

- In addition to the parallelized version, for each application also a functionally equivalent sequential application should be implemented.
- Each application should be explicitly characterized (a) by the number of operations for each process and (b) by the communication demand for each channel.
- For each application, a reference input and output should be included as a separate file.

2.2. Measuring Peak Computational Performance

2.2.1. Matrix Multiplication

The application which was selected as the basis for measuring and comparing the peak performance of several multiprocessor architectures is matrix multiplication (MM). MM is a fundamental, computationally intensive kernel which is utilized in many signal processing applications. It exposes a large degree of parallelism and may be expressed as a synchronous data flow graph, thus meeting the requirements which were set for benchmark applications in 2.1. MM is defined as follows.

The product of two matrices $A \in \mathbb{R}^{M \times N}$ and $B \in \mathbb{R}^{N \times P}$ is a matrix $C \in \mathbb{R}^{M \times P}$, of which the elements are given by:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}$$

for each pair i and j with $1 \leq i \leq$ M and $1 \leq j \leq$ P:

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{bmatrix}$$

In the following sections, the multiplication of square matrices ($A, B, C \in \mathbb{R}^{N \times N}$) is considered. The goal is to specify the MM operation according to the SDF model so that it can be executed efficiently in parallel, in a scalable (in terms of problem size, computation/communication granularity), platform-independent manner.

2.2.2. Previous Implementations

MM has already been implemented as an SDF in two well-known software frameworks as discussed below.

BSC Cell Superscalar

Cell Superscalar (CellSs) [23] is a framework developed at Barcelona Supercomputing Center, which addresses the automatic exploitation of the functional parallelism of a sequential program through the different processing elements of the Cell BE architecture [24]. CellSs suggests a simple programming model for specifying (sequential) applications and uses a runtime library to exploit their existing parallelism by building at runtime a task dependency graph. The runtime environment takes care of the task scheduling and data handling between the different processors of this heterogeneous architecture (CBE is composed of a 64-bit multithreaded PowerPC processor element (PPE) and eight synergistic processor elements (SPEs)).

Among the code examples that are available on the project's website in order to demonstrate the use of CellSs is a MM implementation. Its parallel execution is organised as follows.

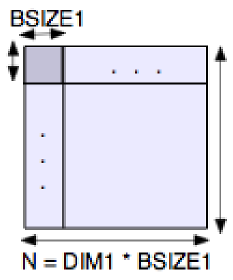


Figure 2.1.: CellSs Matrix Decomposition

Matrices A and B of size $N \times N$ as well as the product matrix C are decomposed into $DIM \cdot DIM$ submatrices (blocks), each of size $BSIZE \times BSIZE$ (where $N = DIM \cdot BSIZE$), as shown in Figure 2.1. The tasks submitted to the Cell SPEs are block multiplications (BM), whereas the scheduling of those tasks and the corresponding data transfers are managed by the PPE, which acts as the control unit in the process network of Figure 2.2. Granularity of block multiplications may be modified through the parameters DIM and BSIZE, which define the number of updates that C blocks should go through before the final product is calculated (hence, the total number of BM operations) and the block size respectively.

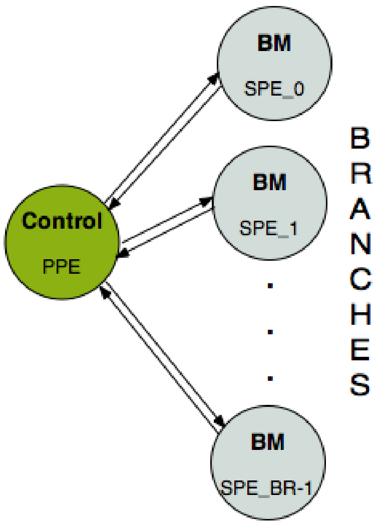


Figure 2.2.: CellSs MM Process Network

The code which is executed on the PPE (BM scheduling) and on the SPEs (BM) of the CBE is shown in the following listings. Communication details have been omitted since data transfers between the PPE and the SPEs are handled by the runtime environment of CellSs. It is assumed that decomposition of matrices A, B and C into $DIM \cdot DIM$ blocks has already taken place before the calculation of product C begins.

```
void compute(A[DIM][DIM], B[DIM][DIM], C[DIM][DIM]){
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      for (k = 0; k < DIM; k++)
        block_addmultiply(A[i, k], B[k, j], C[i, j]);
}
```

Listing 2.1: CellSs Control function

```
void block_addmultiply(Ab[BSIZE][BSIZE], Bb[BSIZE][BSIZE],
                       Cb[BSIZE][BSIZE]){
  for (x = 0; x < BSIZE; x++)
    for (y = 0; y < BSIZE; y++)
      for (k = 0; k < BSIZE; k++)
```

```
            Cb[x, y] += Ab[x, k] * Bb[k, y];
}
```

Listing 2.2: CellSs Block Matrix Multiplication function

Note: Several optimization techniques such as vectorized code, loop unrolling or data prefetching could be applied to maximize the performance of block multiplication execution.

Algorithm Analysis

Computation: For the above described parallel execution of matrix multiplication, a total of $2 \cdot BSIZE^3 \cdot DIM^3 = 2 \cdot N^3$ floating-point operations are required, as the computation of each element of product matrix C involves N additions and N (pair-wise) multiplications (inner product of an A's row and a B's column, see Listing 2.2).

Communication: In terms of communication, the CellSc MM algorithm demands a transfer of $4 \cdot DIM^3 \cdot BSIZE^2$ packets in total between Cell's processor elements. That is because in each of the DIM^3 block multiplications, 4 blocks of size $BSIZE \times BSIZE$ need to be transferred: blocks A[i, k], B[k, j], C[i, j] are sent from the PPE to an SPE and then the updated version of C[i, j] is sent back from the corresponding SPE to the PPE (see Listing 2.1/line 5).

The following graph [23] shows the performance of several versions of the algorithm (when parallelized by CellSs) in comparison with the machine's peak. The six versions of MM only differ in the BM code (ranging from non vectorized to highly optimized). Tests have been run using 2048 × 2048 matrices divided into blocks of 64 × 64 elements ($N = 2048$, $BSIZE = 64$).

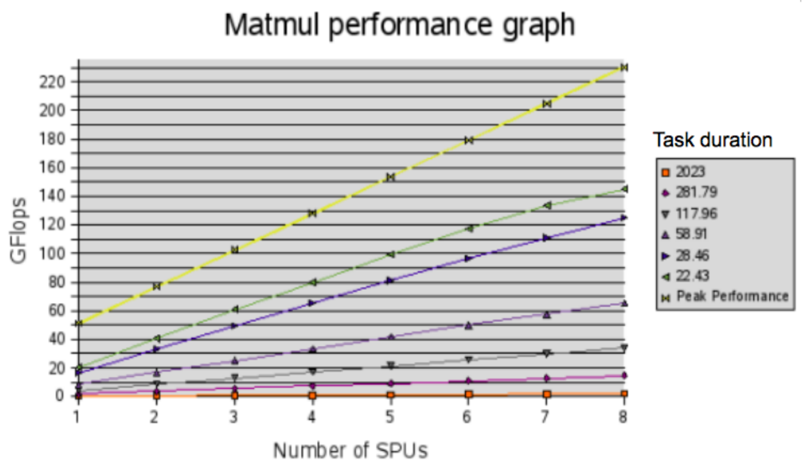


Figure 2.3.: CellSs MM Performance Graph [23]

Note: Theoretical Peak of the CBE is 230.4 GFlops for 8 SPEs.

StreamIt

StreamIt [19] is a programming language and a compilation infrastructure, specifically engineered for modern streaming systems. It is designed to facilitate the programming of large streaming applications, as well as their efficient and effective mapping to a wide variety of target architectures, including commercial-off-the-shelf uniprocessors, multicore architectures and clusters of workstations. A set of benchmark applications for the evaluation of StreamIt compiler is available in [19]. Among those applications, a block algorithm for matrix multiplication has also been implemented.

The specific benchmark (*matmul-block*) generates a series of matrices and multiplies them. In order to reduce the amount of communication, the matrices are divided into equal-sized submatrices, which are reordered, pairwise multiplied, and reordered again to get the final result matrix. More specifically, MM is executed by following the next four steps:

1. Matrix B is transposed.
2. Matrices A and B are reordered and divided into $DIM \cdot DIM$ submatrices (blocks) of size $BSIZE \times BSIZE$.
3. Blocks are multiplied pairwise as described in the C-code of Listing 2.3.
4. Product blocks are reordered internally and recombined in order to form the final product matrix C.

```
void compute(A[DIM][DIM], B[DIM][DIM], C[DIM][DIM]){
  for (k = 0; k < DIM; k++)
    for (j = 0; j < DIM; j++)
      for (i = 0; i < DIM; i++)
        C[j, i] += block_mult(A[k, i], B[j, i]);
}
```

Listing 2.3: StreamIt Block Matrix Multiplication

Based on the description of the above algorithm using the StreamIt programming language (given in [19]), the SDF graph for the execution of block MM can be derived. Figure 2.4 depicts the corresponding process network. In this graph, nodes represent processes and edges represent communication channels between the respective processes. Each edge is annotated with the number of floating-point values which are transferred over the communication channel at each firing of the destination-process (i.e., the process for which the corresponding edge is incoming).

Data flow over this process network can be described as follows.

- 1st and 2nd Phase: Matrix B is transposed within process *Transpose*. Matrix A and the output stream of *Transpose* are reordered and split into DIM^2 blocks, each consisting of $BSIZE^2$ floating-point elements (processes *Split*).

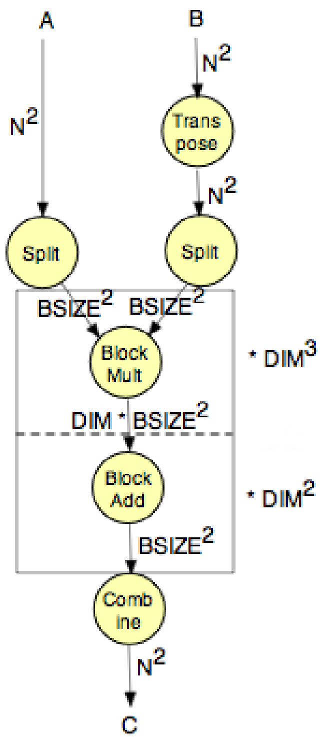


Figure 2.4.: StreamIt MM Process Network

- 3rd Phase (part of the graph within rectangle): Product blocks of matrix C are calculated (Phase 3 is iterated DIM^2 times). Computation of each block C[j, i] (i=[1,...,DIM], j=[1,...,DIM]) involves:
  - DIM firings of process *Block Multiply*, which multiplies blocks A[k, i], B[j, i] (k=[1,...,DIM]). At each firing of *Block Multiply* a total of $2 \cdot BSIZE^2$ elements is read from the output streams of processes *Split* and $BSIZE^2$ elements are buffered in the output stream of *Block Multiply*.
  - 1 firing of process *Block Add*, within which C[j, i] is calculated. After completion of the previous step, $DIM \cdot BSIZE^2$ elements (DIM updates of block C[j, i]) are buffered in the communication channel between *Block Multiply* and *Block Add*. Those elements are consumed when the latter fires. *Block Add* combines the updates of C[j,i] to give the final product block of size $BSIZE^2$.
- 4th Phase: The DIM^2 product blocks are reordered and recombined to form the final result matrix C (process *Combine*).

Algorithm Analysis

Computation: The above described block matrix multiplication algorithm requires a total of $2 \cdot BSIZE^3 \cdot DIM^3 = 2 \cdot N^3$ floating-point operations as any other MM algorithm ($A, B, C \in \mathbb{R}^{N \times N}$).

Communication: Based on data flow analysis for the process network of Figure 2.4, the total amount of transferred data during execution of the algorithm equals $DIM^2 \cdot (3 \cdot DIM + 1) \cdot BSIZE^2$ floating-point values.

Although this version of block MM results in less communication than the CellSs implementation, however mapping the process network to any underlying architecture is not trivial as in the latter case. Therefore, remapping and rescheduling of processes every time the number of processors on a given MPSoC or the whole architecture changes may prove to be rather challenging. Furthermore, StreamIt algorithm is conceptually more complicated and demands more memory accesses than the CellSs alternative due to required reordering and recombination of all involved matrices' blocks at Phases 1, 2 and 4. Those memory operations could contribute a non-negligible overhead to execution time.

Other implementations

Numerous parallel algorithms for MM have been proposed in literature. Among them, the dynamic load-balancing algorithms in [25] and [26], the parallel version of Strassen's algorithm in [27], the scalable universal matrix multiplication algorithm (SUMMA) in [28], the systolic algorithm in [29] and others have been especially designed targeting heterogeneous and/or reconfigurable multiprocesssor platforms. However, most of these approaches are not compatible with the SDF model of computation and hence, do not comply with the specification requirements of DOL.

2.2.3. Algorithm and Implementation

This section describes an algorithm for the parallel (block) execution of matrix multiplication based on the SDF model of computation, which has been implemented on the DOL. It uses the CellSs parallel MM implementation as a starting point, but reduces the communication, while keeping the required computational effort unchanged. Compared to the StreamIt implementation, our parallel MM algorithm leads to reduced data transfers among the processors of the MPSoC architecture. At the same time it is simpler in conception, it needs fewer re-orderings of the matrix elements in memory and no re-combination of the matrix blocks. Moreover, its mapping to any underlying multiprocessor architecture is more trivial, which leads to improved scalability efficiency with regard to the number of available processors.

Two versions of the proposed parallel MM algorithm are presented. In the first one (1-stage), block multiplication (BM) of the A and B submatrices is executed sequentially. On the contrary, in the second version (2-stage), the parallelism of this task is also exploited so as to further reduce the total execution time. For both versions there are two available implementations depending on whether inter-process communication is achieved via simple or windowed FIFO queues.

1-stage Matrix Multiplication

As has already been mentioned, the 1-stage MM algorithm (1sMM) is based on the CellSs implementation, which was presented in the previous section. The process network of Figure 2.2 may be generalized as shown in Figure 2.5 so that it is no longer restricted to the CBE architecture.

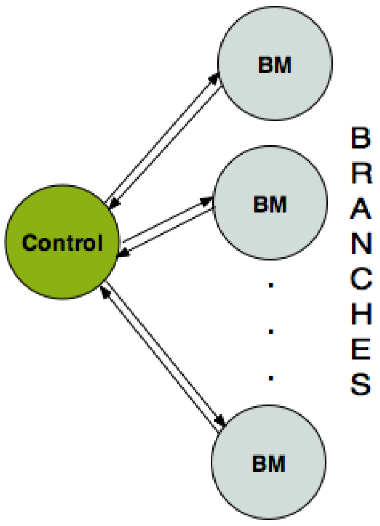


Figure 2.5.: 1sMM Process Network

On any underlying multiprocessor architecture, the *Control* process is mapped on one processor, which will thereafter act as the organizing unit of MM, scheduling the block multiplications, sending the required data to BM processes and updating the product matrix C according to the received intermediate results. The remaining $BRANCHES$ processors are committed to execute block multiplication between the A and B submatrices that they receive from *Control* at each firing.

The parameters that are used to configure the size and granularity of the parallel MM problem as well as the number of processes that will participate in the algorithm execution are summarized in the following table.

Table 2.1.: 1sMM Parameters

Parameter	Explanation
BRANCHES	Number of BM processes
DIM	Number of A, B, C Blocks $(DIM \cdot DIM)$
BSIZE	Block Size $(BSIZE \times BSIZE)$
P	Buffering Factor

It can be observed that the communication demand of the CellSs implementation could be reduced from $4 \cdot DIM^3 \cdot BSIZE^2$ to $DIM^2 \cdot (2 \cdot DIM + 1) \cdot BSIZE^2$

floating-point values if the product blocks are kept in the BM processes instead of being transferred to and from them whenever a block update is needed.

The idea is based on the fact that the computation of each product block of matrix C is assigned to a specific BM process. This process receives a row i of A blocks and a column j of B blocks in DIM consecutive steps, after which it should have computed the final corresponding block C_{ij}. At each step of the CellSs implementation the BM process receives an instance of the targeted C block, which it updates and sends back to the Control unit, where all intermediate results are kept. In order to reduce the communication, each BM process can maintain a local block C_b (initially filled with zero values) and internally aggregate the contribution of A and B submatrices which it receives at each firing. In this case, the BM process needs to transfer data to the Control unit only after DIM steps, when its local block will correspond to the product submatrix C_{ij}. After this transfer, the local block is cleared and the computation of another product block may begin.

Communication in the process network is reduced by almost 50% since for each update of the DIM^2 C submatrices only 2 (instead of 4) blocks of size $BSIZE \times BSIZE$ need to be transferred between the *Control* and a BM process. Moreover, after DIM updates of each C submatrix, one block of equal size is sent from the corresponding BM process to *Control*, thus resulting in a total communication cost of $DIM^2 \cdot (2 \cdot DIM + 1) \cdot BSIZE^2$ floating-point values for the computation of the entire result matrix C.

Given the above mentioned modification to the original CellSs approach, the functionality of the Control and BM units is described in Listing 2.4 and Listing 2.5 respectively.

```
void compute(A[DIM][DIM], B[DIM][DIM], C[DIM][DIM]){
  for (i = 0; i < DIM; i++) {
     for (k = 0; k < DIM / BRANCHES; k++) {
        for (j = 0; j < DIM; j++) {
               for (l = 0; l < BRANCHES; l++){
               send to branch l:
                  A[i,j], B[j, k * BRANCHES + l];
           }
        }
        if (i >= P) {
           for (l = 0; l < BRANCHES; l++) {
               read from branch l:
                  C[i - P, k * BRANCHES + l];
           }
            }
     }
  }
  for(i = DIM - P; i< DIM ; i++){
    for(k = 0; k < DIM / BRANCHES; k++){
      for(l = 0; l < BRANCHES; l++){
         read from branch l:
            C[i, k * BRANCHES + l];
      }
    }
  }
}
```

Listing 2.4: 1sMM Control function

```
void block_addmultiply(Ab[BSIZE][BSIZE], Bb[BSIZE][BSIZE]){
  for (x = 0; x < BSIZE; x++)
    for (y = 0; y < BSIZE; y++)
      for (k = 0; k < BSIZE_V; k++)  // BSIZE_V = BSIZE/4
        for (l = 0; l < 4; l++)
          Cb[4 * (x * BSIZE_V + k) + l] += Ab[x * BSIZE + y] *
                                          Bb[4 * (y * BSIZE_V + k) + l];
}
```

Listing 2.5: 1sMM Block Matrix Multiplication function

As can be seen in the code, *vectorization* has been applied to increase the performance of block multiplication (a vector length of 4 is assumed). Furthermore, the results of BM processes (C product blocks) are buffered in order to enable overlapping of computation and communication during execution of the MM algorithm. *Buffering* is controlled by the parameter P. If P is zero, the results of all BM branches are collected by Control immediately after writing the corresponding A and B submatrices on them (in DIM steps). Otherwise, if P is greater than zero, another $(P-1)$ iterations of block multiplication take place before the results of the BM processes are collected.

Algorithm Analysis

Computation: The computational effort needed for the execution of the 1sMM remains unchanged, therefore equal to $2 \cdot N^3$ floating-point operations in total.

Communication: Communication cost of the 1sMM is $DIM^2 \cdot (2 \cdot DIM + 1) \cdot BSIZE^2$ transferred floating-point values as has already been analysed.

Alternative Implementation

In this subsection, an alternative implementation of the 1sMM algorithm is presented. The main difference compared to the previous implementation regards the decomposition method of the multiplied matrices A, B. Until now we have only considered the decomposition of those matrices into square blocks of size $BSIZE \times BSIZE$ (which is also the size of the product C submatrices) as shown in Figure 2.1. In this version of 1sMM, A and B are divided into row and column blocks respectively so that matrix multiplication is organised according to the following equation:

$$\mathbf{AB} = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_m \end{bmatrix} \begin{bmatrix} B_1 & B_2 & \dots & B_p \end{bmatrix} = \begin{bmatrix} (A_1 \cdot B_1) & (A_1 \cdot B_2) & \dots & (A_1 \cdot B_p) \\ (A_2 \cdot B_1) & (A_2 \cdot B_2) & \dots & (A_2 \cdot B_p) \\ \vdots & \vdots & \ddots & \vdots \\ (A_m \cdot B_1) & (A_m \cdot B_2) & \dots & (A_m \cdot B_p) \end{bmatrix}$$

$$A_i = \begin{bmatrix} a_{i,1} & a_{i,2} & \cdots & a_{i,n} \end{bmatrix}$$
$$B_i = \begin{bmatrix} b_{1,i} & b_{2,i} & \cdots & b_{n,i} \end{bmatrix}^T.$$

Block Matrix Multiplication is executed in this case as presented in Figure 2.6.

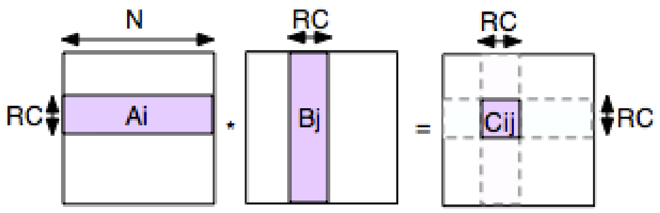


Figure 2.6.: 1sMM Row/Column Block Decomposition

Each row block A consists of RC rows of size N ($A_b \in \mathbb{R}^{RC \times N}$), whereas each column block B is composed of RC columns of size N ($B_b \in \mathbb{R}^{N \times RC}$). Block multiplication in this case results in a product block of size $RC \times RC$ (parameter RC is the equivalent of $BSIZE$ from the first 1sMM version). The parameters which configure the input of this MM alternative are summarized as follows.

Table 2.2.: 1sMM Alternative Parameters

Parameter	Explanation
BRANCHES	Number of BM Processes
DIM	Number of A, B, C Blocks (DIM for A, B, $DIM \cdot DIM$ for C)
RC	Number of Rows per Row-block,
	Number of Columns per Column-block
P	Buffering Factor

Computation of each product block requires no longer DIM successive steps as in the first 1sMM version. On the contrary it is completed in just one step, during which the Control unit sends an A row block and a B column block to the corresponding BM process, which multiplies them and writes the result back to Control. Buffering of results is used so that computation and communication can be overlapped during algorithm execution. The code that describes the modified functionality of the Control and BM processes is included in Listing 2.6 and Listing 2.7.

```
void compute(A[DIM][DIM], B[DIM][DIM], C[DIM][DIM]){
  for (i = 0; i < DIM; i++) {
    for (j = 0; j < DIM / BRANCHES; j++) {
      for (k = 0; k < BRANCHES; k++) {
         send to branch k:
                        A[i], B[j * BRANCHES + k];
      }
      if (i >= P) {
           for (k = 0; k < BRANCHES; k++) {
           read from branch k:
              C[i - P, j * BRANCHES + k];
        }
      }
    }
  }
  for(i = DIM - P; i < DIM; i++){
    for(j = 0; j < DIM / BRANCHES; j++){
      for(k = 0; k < BRANCHES; k++){
         read from branch k:
            C[i, j * BRANCHES + k];
      }
    }
  }
```

Listing 2.6: 1sMM Alternative Control function

```
void block_addmultiply(Ab[BSIZE][BSIZE], Bb[BSIZE][BSIZE]){
  for (i = 0; i < RC; i++)
    for (j = 0; j < RC; j++)
      for (k = 0; k < N; k++)
          Cb[i * RC + j] += Ab[i * N + k] * Bb[j * N + k];
}
```

Listing 2.7: 1sMM Alternative Block Matrix Multiplication function

Algorithm Analysis

Computation: The overall computational effort needed for MM remains equal to $2 \cdot N^3$ floating-point operations.

Communication: Communication cost equals $DIM^2 \cdot (2 \cdot RC \cdot N + RC^2)$ transferred floating-point values, which is practically the same as in the first 1sMM version, given that $N = RC \cdot DIM$ and $RC \equiv BSIZE$.

The disadvantage of this technique, however, is that the size of transferred data (row and column block) between Control and a BM process at each firing of the latter is proportional to the matrix dimension N. That could pose the demand for unacceptably wide communication channels (FIFOs), which may not be available on certain architectures, or could also violate the restrictions in local memory (capacity) of processes, where blocks should be maintained while being processed.

2-stage Matrix Multiplication

In this version of parallel MM, the 1-stage algorithm is extended so that execution of block multiplication on each BM process can be further parallelised. As in the 1-stage algorithm, the matrices A and B are initially decomposed into $DIM1 \cdot DIM1$ square blocks of size $BSIZE1 \times BSIZE1$. Each BM process receives $2 \cdot DIM1$ blocks (in $DIM1$ consecutive steps) which it multiplies pairwise internally in order to compute one block of the final product matrix. In the MM implementations that have been presented so far, parallelism of the BM task has not been exploited. Nevertheless, the 1-stage algorithm could be recursively applied at this level of computation to accelerate the execution of BM on condition that the processors on which BM processes are mapped contain more than one computational cores (otherwise, 2-stage MM is equivalent to initial algorithm). Recursive execution of the 1sMM within the BM task means that the received blocks of size $BSIZE1 \times BSIZE1$ are further decomposed into $DIM2 \cdot DIM2$ sub-blocks of size $BSIZE2 \times BSIZE2$ so that the elements of the product block can be calculated in parallel (with each product sub-block being calculated in $DIM2$ steps). The process network for the proposed 2-stage parallel implementation of MM as well as the structure of matrices, blocks and sub-blocks at the different levels of execution are presented in the following figures.

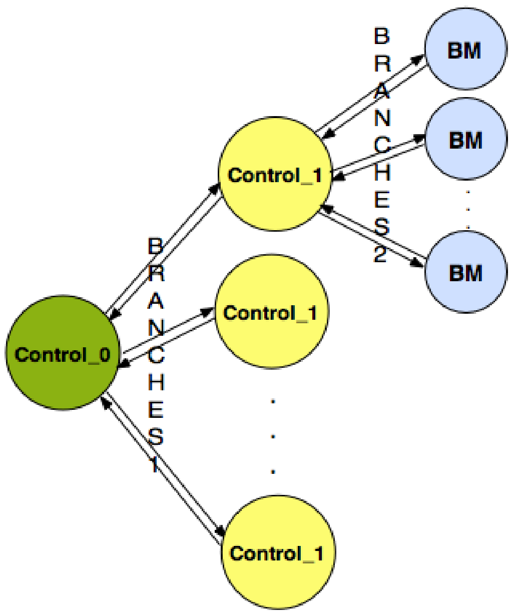


Figure 2.7.: 2sMM Process Network

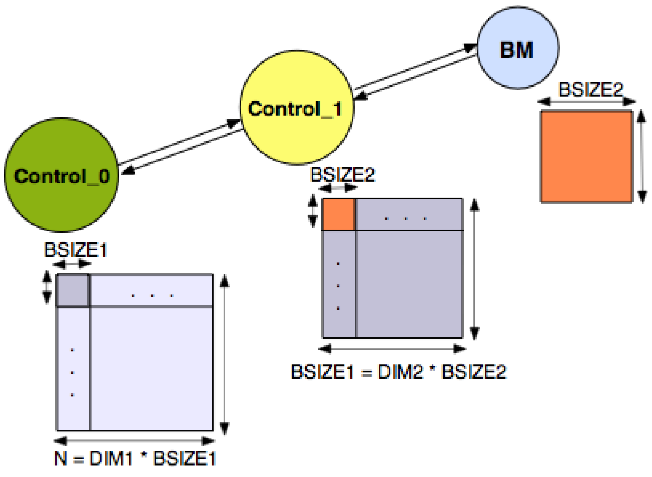


Figure 2.8.: 2sMM Matrix Decomposition

Process *Control_0* orchestrates MM. It decomposes the multiplied matrices, sends the blocks to corresponding *Control_1* processes and every $DIM1$ steps, it receives one product block from each of the $BRANCHES1$ branches. Likewise, each *Control_1* unit decomposes received blocks and forwards the sub-blocks further to BM processes which multiply them pairwise and return after $DIM2$ steps the corresponding product sub-blocks.

All parameters of the 2-stage MM algorithm are summarized in Table 2.3.

Table 2.3.: 2sMM Parameters

Parameter	Explanation
BRANCHES1	Number of Control_1 Processes
BRANCHES2	Number of BM Processes
DIM1	Number of A, B, C Blocks on the 1st Level ($DIM1 \cdot DIM1$)
DIM2	Number of A, B, C Blocks on the 2st Level ($DIM2 \cdot DIM2$)
BSIZE1	Block Size on the 1st level ($BSIZE1 \times BSIZE1$)
BSIZE1	Block Size on the 2st level ($BSIZE2 \times BSIZE2$)
P1	Buffering Factor on the 1st level
P2	Buffering Factor on the 2nd level

The C code which defines the functionality of processes Control_0, Control_1 and BM is included in the following listings.

```
void compute_0(A[DIM1][DIM1], B[DIM1][DIM1], C[DIM1][DIM1]){
  for (i = 0; i < DIM1; i++) {
     for (k = 0; k < DIM1 / BRANCHES1; k++) {
        for (j = 0; j < DIM1; j++) {
               for (l = 0; l < BRANCHES1; l++){
               send to branch l:
                  A[i,j], B[j, k * BRANCHES1 + l];
            }
        }
        if (i >= P1) {
           for (l = 0; l < BRANCHES1; l++) {
               read from branch l:
                  C[i - P1, k * BRANCHES1 + l];
           }
            }
     }
  }
  for(i = DIM1 - P1; i< DIM1 ; i++){
    for(k = 0; k < DIM1 / BRANCHES1; k++){
       for(l = 0; l < BRANCHES1; l++){
          read from branch l:
             C[i, k * BRANCHES1 + l];
       }
    }
  }
}
```

Listing 2.8: 2sMM Control function (1st level)

```
void compute_1(Ab[BSIZE1][BSIZE1], Bb[BSIZE1][BSIZE1]){
  decompose blocks Ab, Bb, local Cb into DIM2 * DIM2 subblocks
      (matrices As[DIM2][DIM2], Bs[DIM2][DIM2], Cs[DIM2][DIM2])

  for (i = 0; i < DIM2; i++) {
     for (k = 0; k < DIM2 / BRANCHES2; k++) {
        for (j = 0; j < DIM2; j++) {
               for (l = 0; l < BRANCHES2; l++){
               send to branch l:
                  As[i,j], Bs[j, k * BRANCHES1 + l];
            }
        }
13          if (i >= P2) {
14             for (l = 0; l < BRANCHES2; l++) {
15                 read from branch l:
16                    Cs[i - P2, k * BRANCHES2 + l];
17             }
18              }
19       }
20    }
21    for(i = DIM2 - P2; i< DIM2 ; i++){
22      for(k = 0; k < DIM2 / BRANCHES2; k++){
23         for(l = 0; l < BRANCHES2; l++){
24            read from branch l:
25               Cs[i, k * BRANCHES2 + l];
26         }
27      }
28    }
29
30    Cb_local += Cs;
31    return Cb_local;
32 }
```

Listing 2.9: 2sMM Control function (2nd level)

```
void block_addmultiply(Asb[BSIZE2][BSIZE2], Bsb[BSIZE2][BSIZE2]){
  for (x = 0; x < BSIZE2; x++)
    for (y = 0; y < BSIZE2; y++)
      for (k = 0; k < BSIZE_V; k++)  // BSIZE_V = BSIZE2/4
        for (l = 0; l < 4; l++)
          Csb[4 * (x * BSIZE_V + k) + l] += Asb[x * BSIZE2 + y] *
                                              Bsb[4 * (y * BSIZE_V + k) + l];
  return Csb;
}
```

Listing 2.10: 2sMM Block Matrix Multiplication function

Buffering of results may be applied on both levels of communication (Control_1 to Control_0 and BM to Control_1). It is controlled by parameters $P1$ and $P2$ respectively.

As can be seen in the C-code of process Control_1 (Listing 2.9, line 30), aggregation of intermediate results for every product block is required through the $DIM1$ steps of its computation. This aggregation creates an overhead of $DIM1 \cdot BSIZE1^2$ floating-point operations (additions) for the calculation of each product block, thus increasing the overall computational cost of MM by a factor of $DIM1^2 \cdot (DIM1 \cdot BSIZE1^2) = N^2 \cdot DIM1$ operations. Aggregation of intermediate results in Control_1 (and consequently, the additional computation overhead) could be avoided in the following cases:

- All $DIM1$ intermediate instances of each product block are written back to process Control_0 instead of only the final one. Such a decision, however, would lead to a significant increase in the size of transferred data between Control_0 and Control_1 (see CellSs implementation).
- Parameter $DIM2$ is restricted to be equal to $BRANCHES2$ so that intermediate instances of the product sub-blocks (accordingly, blocks) can be maintained within the BM processes, thus eliminating the need for aggregation within Control_1 processes. That restriction, however, would limit the degree of freedom in parametrization of the 2-stage MM algorithm, since the number of sub-blocks into which blocks are decomposed

within Control_1 would be a function of the number of available processing elements on which BM can be executed.

Therefore, there seems to be a trade-off between the computational overhead of aggregating intermediate results in Control_1 (and its possible impact on total execution time of MM) and the communication cost between Control_0 and Control_1 or the degree of parallelization which can be applied during execution of block multiplication. The 2-stage MM implementation could be adapted appropriately to achieve optimal performance over different architectures.

Algorithm Analysis

Computation: Aggregation of intermediate results in Control_1 processes increases MM computation cost from $2 \cdot N^3$ to $(2 \cdot N^3 + N^2 \cdot DIM1)$ floating-point operations. This overhead is not negligible and could possibly overshadow the gain in execution time that is achieved due to parallel implementation of BM.

Communication: The size of transferred data over the two levels of communication channels is equal to $DIM1^2 \cdot [(2 \cdot DIM1 + 1) \cdot BSIZE1^2 + DIM2^2 \cdot (2 \cdot DIM2 + 1) \cdot BSIZE2^2]$ floating-point values.

2.3. Measuring Peak Communication Bandwidth

The application which is proposed for measuring the peak communication bandwidth of any given heterogeneous MPSoC with distributed-memory architecture is the trivial communication-dominated generator-consumer application which is described in the following.

2.3.1. Generator-Consumer Application

The process network of the generator-consumer application is depicted in Figure 2.9. It consists a simple pipeline, in which process *Generator* generates constantly data, which are forwarded through a chain of processes to destination-process *Consumer*. The size of data which is received (buffered) by each intermediate process and forwarded to its neighbour at every firing can be defined so that communication over the process network reaches its limits, i.e. capacity of available communication channels is fully exploited and maximum data throughput is achieved.

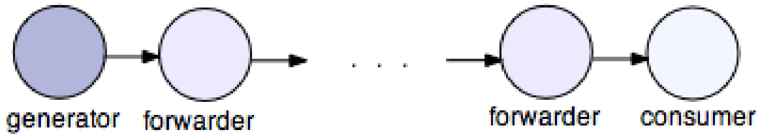


Figure 2.9.: Generator - Consumer Process Network

Algorithm Analysis

Computation: No data processing takes place within the intermediate (*forwarding*) processes of the chain or *Consumer*. Therefore, the overall computational cost of the Generator-Consumer application is equal to the overhead of producing the data to be transmitted within *Generator*.

Communication: Communication cost of this application depends directly on the size of tokens S which are transmitted over the process network as well as the number of intermediate (forwarding) processes F. At each firing of the processes, S bytes are transferred over each of the totally $(F + 1)$ channels. Therefore, the aggregate bandwidth of communication is equal to $(F + 1) \cdot S$ bytes at each firing of the chain processes.

2.4. Summary

In this chapter, two benchmark applications have been proposed for the evaluation of the peak computational performance and the peak communication bandwidth, respectively, of various multiprocessor architectures. The computation and communication cost of all presented versions of matrix multiplication and the generator-consumer application are summarized in the following table. The parameters are in accordance with the ones specified in the preceding sections.

Table 2.4.: Computation and communication cost of presented algorithms

Algorithm	Computation	Communication
CellSs MM	$2 \cdot N^3$	$4 \cdot DIM^3 \cdot BSIZE^2$
StreamIt MM	$2 \cdot N^3$	$DIM^2 \cdot (3 \cdot DIM + 1) \cdot BSIZE^2$
1-stage MM (square blocks)	$2 \cdot N^3$	$DIM^2 \cdot (2 \cdot DIM + 1) \cdot BSIZE^2$
1-stage MM (row/column blocks)	$2 \cdot N^3$	$DIM^2 \cdot (2 \cdot RC \cdot N + RC^2)$
2-stage MM	$2 \cdot N^3 + N^2 \cdot DIM1$	$DIM1^2 \cdot [(2 \cdot DIM1 + 1) \cdot BSIZE1^2$
		$+DIM2^2 \cdot (2 \cdot DIM2 + 1) \cdot BSIZE2^2]$
Generator-consumer pipeline	–	$(F + 1) \cdot S$

3 Cell Broadband Engine

3.1. DOL Run-Time Environment for the CBE

The distributed operation layer contains a code generation back-end that allows the efficient execution of applications, which are specified as Kahn process networks, on the Cell Broadband Engine [30]. The code generation back-end relies on a lightweight run-time system based on protothreads and windowed FIFOs:

– **Protothreads** are usually used for programming constrained (in terms of memory and performance) embedded systems, such as wireless sensor nodes. Protothreads are a simple, yet effective, approach to execute preemptive processes using a single CPU context and a single stack. Therefore, the context switch overhead is very low and no further multi-threading support is required to execute multiple processes on a single processor.

– Unlike standard FIFOs, **windowed FIFOs** support direct access to a continuous data segment in the (circular) FIFO buffer. These segments are called "windows" which leads to the name "windowed FIFO". Compared to standard FIFOs, windowed FIFOs are more efficient because unnecessary memory copies can be avoided. The Kahn process network semantics is not affected by using windowed FIFOs instead of standard FIFOs.

The main features of the run-time system are:

- cooperative multi-threading on the PPE and the SPEs,
- direct windowed FIFO communication between processes mapped to SPEs (PPE not involved) and
- overlapping of computation and communication by making use of DMA engines (memory flow controllers).

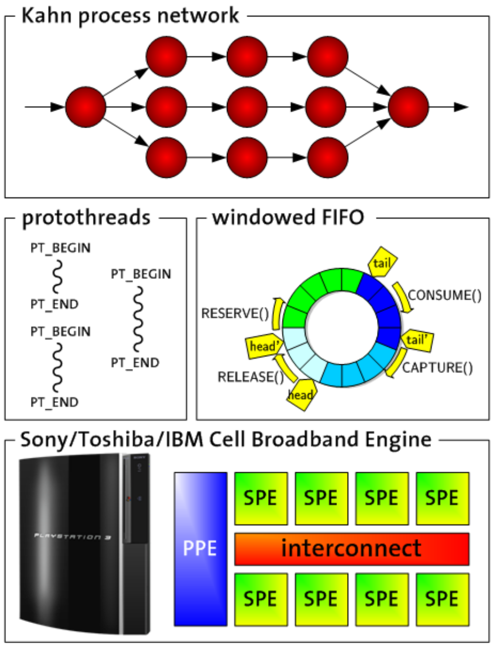


Figure 3.1.: Execution of Kahn Process Networks on CBE

These characteristics enable an efficient, completely distributed execution of Kahn process networks on the CBE.

As has been mentioned in Chapter 1, one of the goals of this thesis was the improvement of the CBE run-time environment. More specifically, in the initial version of it, although the thread and FIFO implementations for on-processor communication were very efficient, the (windowed) FIFO implementation for inter-processor communication did not always work correctly and it yielded lower peak transfer rates than other communication schemata in run-time systems designed for the CBE.

The main objective, therefore, was to re-examine and modify the DOL code generation back-end for the CBE appropriately so as to finally obtain a reliable platform for experiments. To achieve this, the steps listed below were followed:

1. Error handling: To assist the process of debugging, the existing implementation was extended with code aimed to catch and handle potential errors (e.g., memory allocation failures) during execution time.

2. Memory management: Handling of dynamic memory operations with regard to the windowed FIFO implementation for inter-processor communication was enhanced to prevent unexpected behaviour of processes during execution time.

3. Size of communication channels: The initial restriction based on which all (standard and windowed) FIFO channels were set to a default size (1024 bytes) was withdrawn. The application programmer may now define the size of each channel separately through the process network XML specification.

Based on the above modifications, the correct execution of Kahn process networks on the CBE is assured, thus enabling the execution of our benchmarks, as described in the following section.

3.2. Experimental Results

The benchmark applications, which were presented in Chapter 2 and were specified and simulated within the DOL framework, are used to evaluate the performance (peak computational performance and communication bandwidth) of the developed run-time environment for the CBE. A Sony PlayStation 3 running Yellow Dog Linux 6.1 has been used for the experiments. The Sony PlayStation 3 contains a single CBE of which the PPE and six SPEs are available for user applications. The GCC compilers PPU-G++ 4.1.1 and SPU-G++ 4.1.1 were used with all optimizations enabled to compile the CBE-specific code that was produced by the DOL front-end.

3.2.1. Matrix Multiplication

Among the available parallel implementations of matrix multiplication, 1sMM (1-stage multiplication with square block decomposition) has been chosen to evaluate the peak computational performance of the targeted run-time environment and the underlying MPSoC. The corresponding process network (Figure 2.5) can be easily mapped to the CBE architecture. The 'organising' *Control* process will run on the PPE, whereas every available SPE can be utilized to execute block multiplications. The number of available SPEs ranges from one to six, hence a speed-up of up to six can be expected when MM is executed in parallel on the given platform.

In the following experiments, three alternative versions of 1sMM are tested. Their difference lies in the code efficiency of the BM task. The first version does not apply any optimization, relying only on the compiler-level optimizations for the efficient execution of BM. In the second one, vectorized code is used (given that 4 floating-point operations can be executed in one cycle on the PPE/SPEs), which is extended with explicit loop unrolling and data prefetching in the last version. In all 1sMM implementations, inter-process communication is implemented using either standard or windowed FIFO queues. All presented results have been averaged over five executions of the corresponding tests. Parameters of execution in tables and graphs are in accordance with the ones defined in Table 2.1.

Since our goal is to measure the peak computational performance in terms of floating-point operations per second, a sufficiently large block size should be selected, so that execution of the MM application reaches its computational limits. As a first step, the performance of 1sMM is evaluated with regard to the size of blocks into which the multiplied matrices are decomposed. Table 3.1 contains the results of 1sMM execution when all suggested code-level optimizations are applied, FIFO queues are used for inter-process communication and the block size ranges from 16×16 to 64×64 floating-point values (from 1 to 16 Kbytes respectively). The limit of 64×64 elements for the block size is derived by the maximum allowed size of a single DMA transfer on the CBE, which is equal to 16384 bytes (at each DMA transfer between main memory and a SPE's local storage one block is transmitted 'over' the FIFO). The matrix size is equal to 1920×1920 (except for the last test) and tests have been run for the cases when either one or four SPEs are available. Execution time is reported in milliseconds and performance of the 1sMM execution is expressed in GFLOPS.

Table 3.1.: MM with blocks of varying size (FIFO communication, vectorization, loop unrolling, data prefetching)

SPEs	DIM	BSIZE	N	Execution Time	GFLOPS
1	120	16	1920	9640	1.47
	80	24	1920	5950	2.38
	60	32	1920	5980	2.37
	48	40	1920	4990	2.84
	40	48	1920	4730	2.99
	32	64	2048	5320	3.23
4	120	16	1920	5340	2.65
	80	24	1920	3370	4.20
	60	32	1920	2280	6.21
	48	40	1920	1890	7.49
	40	48	1920	1840	7.69
	32	64	2048	1560	11.01

As expected, the computational performance during 1sMM execution (as well as the achieved speed-up when four SPEs run BM in parallel) improves as the block size increases. Therefore, in order to assess the peak performance of the targeted platform under the 1sMM scenario, more experiments are conducted with matrices of size 2048×2048 (1920×1920 in some cases due to restriction that DIM must be a multiple of BRANCHES) divided in blocks of 64×64 elements. Execution results (elapsed time for computation of product matrix in milliseconds, achieved speed-up per matrix element, performance in GFLOPS) for all alternative versions of 1sMM for both FIFO- and windowed FIFO-based communication are included in the following tables.

Table 3.2.: MM with blocks of 64×64 elements (no optimizations)

			FIFO			WFIFO		
SPEs	DIM	N	Exec.time	Speedup	GFLOPS	Exec.time	Speedup	GFLOPS
1	32	2048	67880	1	0.25	66130	1	0.26
2	32	2048	34310	1.98	0.50	33420	1.98	0.51
3	30	1920	18870	3.16	0.75	18380	3.16	0.77
4	32	2048	17200	3.95	1.00	16750	3.95	1.03
5	30	1920	11350	5.26	1.25	11060	5.26	1.28
6	30	1920	9460	6.31	1.5	9230	6.3	1.53

Table 3.3.: MM with blocks of 64 × 64 elements (vectorization)

			FIFO			WFIFO		
SPEs	DIM	N	Exec.time	Speedup	GFLOPS	Exec.time	Speedup	GFLOPS
1	32	2048	10590	1	1.62	10450	1	1.64
2	32	2048	5670	1.87	3.03	5590	1.87	3.07
3	30	1920	3140	2.96	4.51	3100	2.96	4.57
4	32	2048	2880	3.68	5.97	2830	3.69	6.07
5	30	1920	1910	4.87	7.41	1890	4.86	7.49
6	30	1920	1610	5.78	8.79	1580	5.81	8.96

Table 3.4.: MM with blocks of 64 × 64 elements (vectorization, loop unrolling, data prefetching)

			FIFO			WFIFO		
SPEs	DIM	N	Exec.time	Speedup	GFLOPS	Exec.time	Speedup	GFLOPS
1	32	2048	5320	1	3.23	5180	1	3.32
2	32	2048	3050	1.74	5.63	2960	1.75	5.8
3	30	1920	1690	2.77	8.38	1650	2.76	8.58
4	32	2048	1560	3.41	11.01	1520	3.41	11.3
5	30	1920	1190	3.93	11.90	1180	3.86	12
6	30	1920	1210	3.86	11.70	1180	3.86	12

As can be seen in the results, the execution time of 1sMM and the maximum attained performance are improved by 1% to 3% when windowed, instead of standard, FIFOs are used for inter-process communication, while the speed-up per computed element is only slightly different between the two implementations. The performance gain achieved by replacing standard with windowed FIFOs would have been even higher if the BM processes shared the same memory. We know that utilization of windowed FIFOs is particularly advantageous in this case since unnecessary copying of data can be completely avoided by directly accessing the FIFO channel buffer. Therefore, in distributed memory architectures, processes executing on a single processor (sharing the same local memory) or in shared memory architectures, processes executing on different processors (having access to a global memory) can profit significantly from this implementation. In our MM process network, however, all processes are mapped to different processors of the CBE (no shared memory), thus limiting the gain from using windowed FIFOs for communication. Of course, the fact that the absolute execution time for accesses to windowed FIFOs is shorter than the time needed for accesses to standard FIFOs [30] is not negligible, especially in communication-intensive applications, such as MM (with frequent FIFO accesses), as indicated by the obtained results.

The speed-up of the discussed 1sMM configuration as well as the improvement in computational performance (GFLOPS) when the number of available SPEs increases from one to six are depicted in Figure 3.2 and Figure 3.3 respectively. Only results for the windowed FIFO implementations are shown, since those from the corresponding FIFO implementations follow similar trends.

Based on these graphs, it can be observed that performance of the two first 1sMM versions (no optimizations / vectorized BM code) scales almost linearly with regard to the number of processors participating in the MM. However,

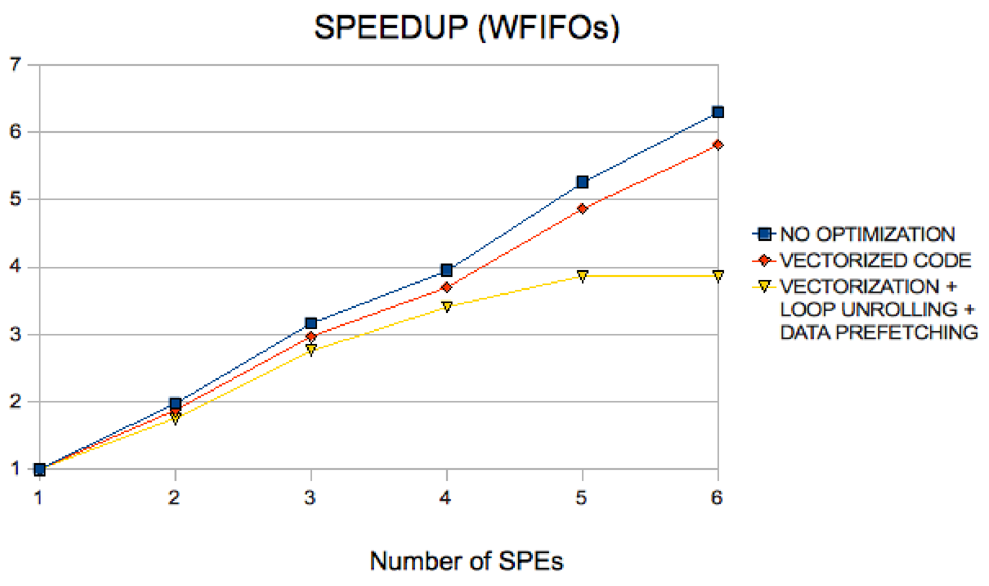


Figure 3.2.: Speed-up diagram for WFIFO MM implementation (BSIZE = 64)

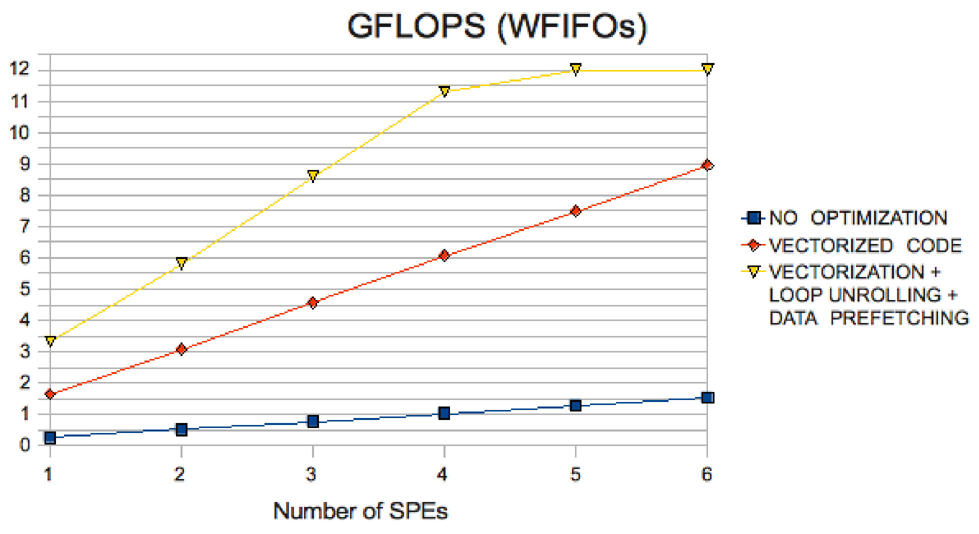


Figure 3.3.: Performance diagram for WFIFO MM implementation (BSIZE = 64)

performance of the third, most refined in terms of code-level optimizations, 1sMM alternative does not scale satisfyingly beyond four SPEs. When five SPEs execute BM in parallel, computational performance reaches its peak (12 GFLOPS) which cannot be surpassed even if more SPEs are used. From that turning point (four SPEs), execution of the specific 1sMM implementation becomes communication-bounded. It has been verified through experiments that for the given implementation and matrices, even if no computation took place within the BM processes (on SPEs), the execution time would practically not be affected. Thus we can assume that beyond this point, inter-process communication is the bottleneck of execution, due to which no further improvement in computational performance is feasible. Even though computation of product blocks takes almost zero time, the actual peak performance of the MPSoC cannot be achieved because of the heavy communication overhead.

That ”turning point”, beyond which performance can no longer scale linearly with the number of processes / processors, would have been reached even earlier if the block size in the 1sMM implementations had been selected to be smaller than 64×64. In this case, the computation-to-communication ratio for the BM processes would be lower, so execution time could become from an early point (small number of running SPEs) dominated by costly inter-process communication operations. To validate this assumption, more tests have been conducted with matrices of size 1024×1024 (960×960 in some cases) divided in blocks of 32×32 elements. The resulting execution speed-up and performance improvement when the number of available SPEs increases for the FIFO implementations (windowed FIFO implementations follow similar trends) is shown in Figure 3.4 and Figure 3.5 respectively. Based on these, when the block size is 32×32, the maximum achieved performance on the CBE fails to scale well beyond 4 SPEs for the second 1sMM version and already beyond 2 SPEs for the third one.

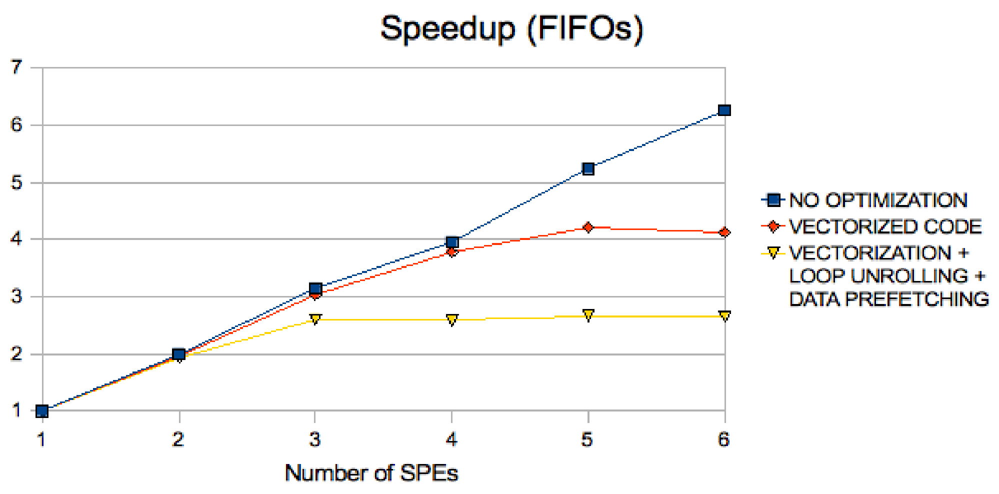


Figure 3.4.: Speed-up diagram for FIFO MM implementation (BSIZE = 32)

Another optimization that could be applied to the BM code is *buffering* of intermediate results, as has been described in Section 2.2. It has been shown through experiments that buffering by a parameter P of 1 or 2 can slightly improve the achieved computational performance (up to 1.5%) in several test cases.

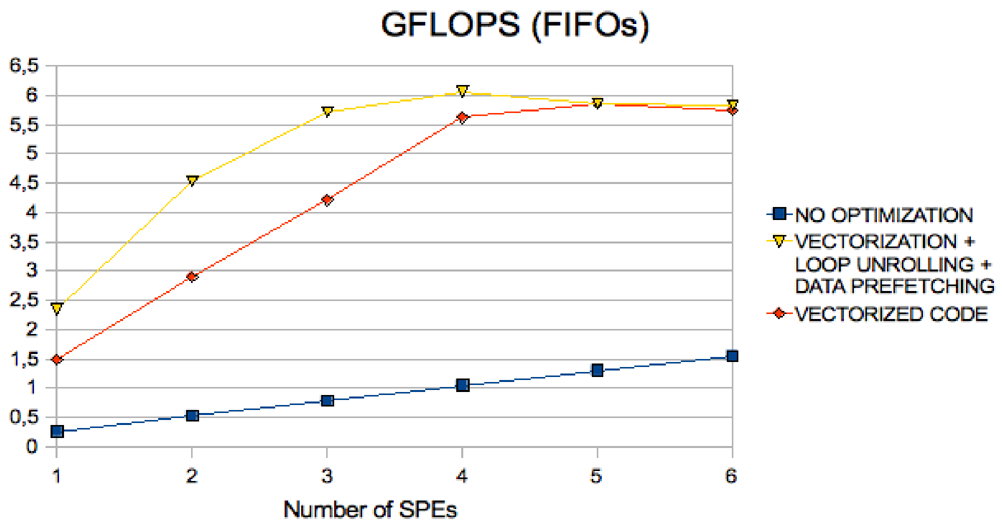


Figure 3.5.: Performance diagram for FIFO MM implementation (BSIZE = 32)

3.2.2. Generator-Consumer

To measure the peak aggregate communication bandwidth over the CBE of PlayStation 3, two experiments were performed. First, a chain of six processes (generator, four forwarding elements, consumer) was mapped on the six available SPEs (one process per SPE). Second, a chain of twelve processes (generator, ten forwarding elements, consumer) was mapped on the six SPEs (two processes per SPE). To connect the processes in the chains, windowed FIFO channels with a size of 16384 bytes were used (maximum allowed size of a single DMA transfer on the CBE).

Table 3.5 summarizes the obtained results (elapsed execution time for 1000000 data transfers over the FIFO queues, transfer rate in Gbytes/s) for the two process chains when the number of bytes transmitted in a single windowed FIFO access ranges from 1024 bytes to 16384 bytes. Figure 3.6 depicts the aggregate inter-SPE data transfer rates for both test cases. The observed peak data rates are 9.87 Gbytes/s when one process (either producer or forwarding element or consumer) is executed on each SPE and 10.91 Gbytes/s when two processes are executed on each SPE. In the latter case the transfer rate is higher because the data transfers initiated by the two processes on each SPE can be partially overlapped [30].

Table 3.5.: Generator-Consumer: Data transfer rate

Processes	Token size (bytes)	Transferred data (Gbytes)	Execution Time (sec)	Transfer rate (Gbytes/s)
6	1024	4.77	0.92	5.18
	2048	9.54	1.61	5.94
	4096	19.07	3.05	6.25
	8192	38.15	5.15	7.41
	16384	76.29	7.73	9.87
12	1024	9.54	1.7	5.61
	2048	19.07	2.8	6.81
	4096	38.15	5.06	7.54
	8192	76.29	9.56	7.98
	16384	152.59	15.38	9.92

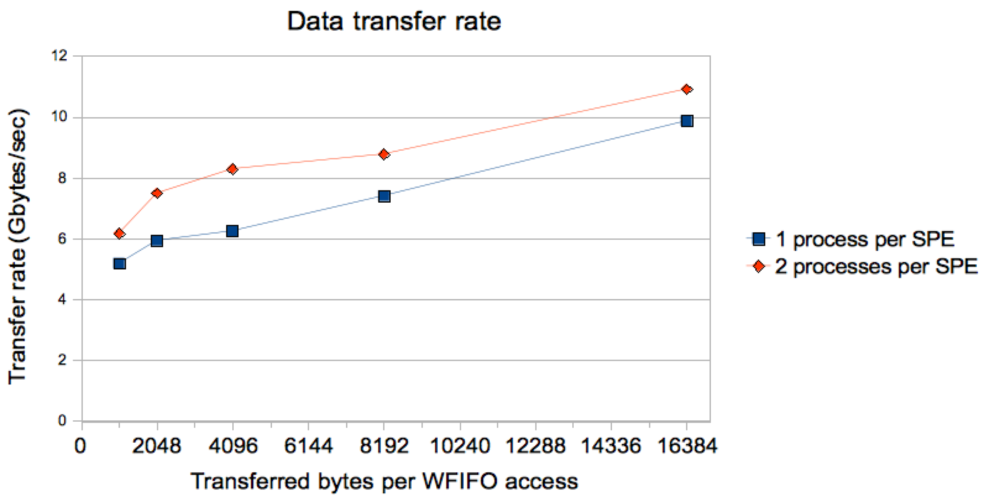


Figure 3.6.: Aggregate inter-SPE data transfer rate

3.3. Summary

The performance of the benchmark applications, which were described in Chapter 2, has been evaluated on the Cell Broadband Engine (Playstation 3). The maximum computation performance and the peak aggregate communication bandwidth that were measured during their execution are 12 GFLOPS and 10.91 Gbytes/s, respectively. The benchmark execution on the CBE has proven rather useful for the enhancement of the DOL run-time environment for the specific multiprocessor architecture.

4 Conclusion

In this semester thesis, a set of benchmark applications has been built, targeting heterogeneous MPSoCs with distributed memory architectures where inter-processor communication is achieved via message passing. The design of these benchmarks has been driven by the need to specify and compare the following properties of different MPSoCs and the run-time environments executing on top of them: (a) peak computational performance in terms of floating-point operations per second and (b) peak (aggregate) bandwidth of interprocessor communication. To this end, several parallel implementations of matrix multiplication as well as a simple, communication-intensive generator-consumer application have been proposed.

The benchmark applications have been specified in accordance with the synchronous data flow model of computation and have been implemented within the DOL software-development framework in a platform-independent, parametrized manner such that they can efficiently execute on several MPSoC platforms. In the corresponding process networks, it is assumed that communication among processes is managed through FIFO or windowed FIFO channels.

As a subsequent step, it has been attempted to evaluate the performance of the developed benchmarks on the Sony/Toshiba/IBM Cell Broadband Engine. To achieve this, the existing DOL code generation back-end for the CBE had first to be revisited and improved so as to obtain a reliable run-time environment for the execution of DOL applications on the targeted architecture. After successful completion of that step, extensive experiments were executed on a PlayStation 3 platform running Yellow Dog Linux 6.1 in order to characterize the performance of the proposed applications on the CBE.

The maximum computational performance has been achieved for the MM test case of 1920×1920 matrices, divided into blocks of 64×64 floating-point elements, when inter-process communication was handled by windowed FIFO queues, all six available SPEs were used and several optimizations (vectorized code, loop unrolling, data prefetching) were applied to the block multiplication task code. In this case, performance of the CBE reached its peak at **12 GFLOPS**, which is however far from the theoretical limit (of approximately 180 GFLOPS for six SPEs), due to a bottleneck caused by inter-process communication. On the other hand, the maximum aggregate bandwidth of inter-SPE communication has been observed during execution of the generator-consumer application when a chain of twelve processes was mapped to the six SPEs of the PlayStation 3 and windowed FIFOs were used for the repeated transfer of 16-Kbyte tokens. In this case, the aggregate data transfer rate was equal to **10.91 Gbytes/s**.

Several directions could be explored for the extension of the work presented in this thesis. The following ones could serve as a starting point for future improvements:

1. Design and development of new benchmark applications: More applications, aimed especially at measuring peak computational performance, could be developed within the DOL framework to characterize the behavior of MPSoCs under different execution scenarios. Emphasis should be put on the equal distribution of workload among parallel-executing processes and the computation-to-communication ratio of them. The latter should be kept as high as possible, so that execution does not get dominated by communication operations and the computational capabilities of the underlying architectures may be at most exploited.

2. Experiments on alternative MPSoC platforms: More tests based on the available or new benchmark applications could be run on other MPSoCs (besides the CBE) to assess and compare their computational and communication capabilities as well as the efficiency of the corresponding run-time environments.

3. Improvement of communication protocol for the CBE run-time environment: The current windowed FIFO implementation for inter-process communication achieves considerably less throughput than other reported implementations. A new, simpler communication protocol could be designed in order to confront this weakness and relieve application execution from communication bottlenecks.

A
Abbreviations

1sMM:	1-stage Matrix Multiplication
2sMM:	2-stage Matrix Multiplication
BM:	Block Multiplication
BSC:	Barcelona Supercomputing Center
CBE:	Cell Broadband Engine
CellSs:	Cell Superscalar
DOL:	Distributed Operation Layer
GFLOPS:	Giga FLoating-point OPerations per Second
KPN:	Kahn Process Network
MM:	Matrix Multiplication
MPSoC:	Multiprocessor System-on-Chip
PPE:	PowerPC Processor Element
SDF:	Synchronous Data Flow
SHAPES:	Scalable Software/Hardware Architecture Platform for Embedded Systems
SPE:	Synergistic Processor Element
XML:	Extensible Markup Language

B
Presentation Slides

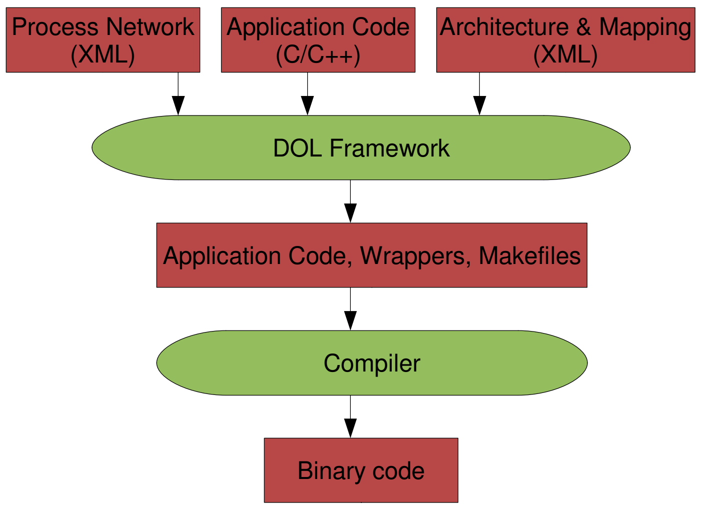
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SHAPES @ TIK

Development of a Streaming Application Benchmark Set

Student: Georgia Giannopoulou
Advisors: Wolfgang Haid, Kai Huang

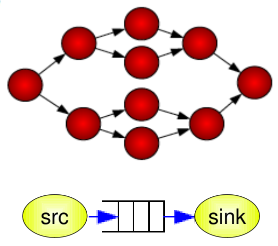
SHAPES - Distributed Operation Layer (DOL)



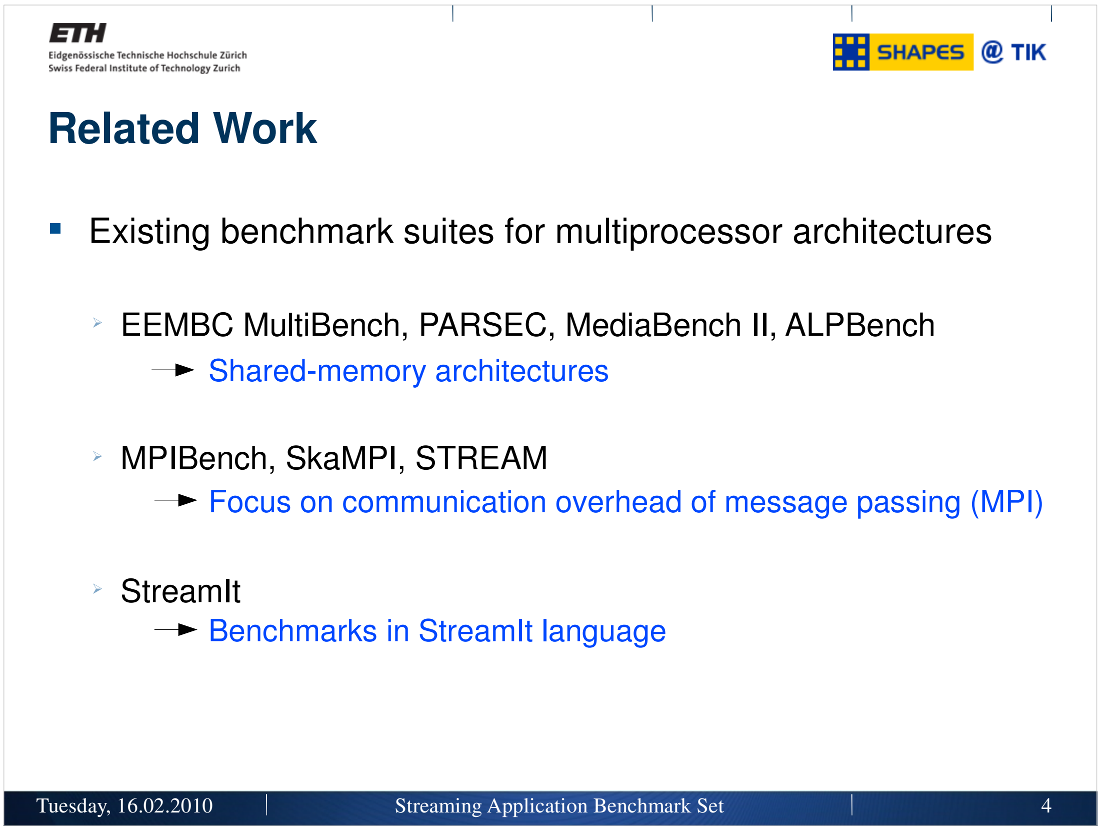
- Specification of applications as Kahn Process Networks
- Efficient mapping and code generation for several multiprocessor architectures

Motivation

- Benchmark development for heterogeneous MPSoC
  - Peak computational performance
  - Peak bandwidth of inter-processor communication (message passing)
- Requirements
  - Synchronous Dataflow Model
  - Platform independence
  - Scalability
  - Parallelism
  - Communication restrictions



- Targeted architectures
  - IBM/Toshiba/Sony Cell Broadband Engine
  - Atmel Diopsis 940 (SHAPES)
  - MPARM

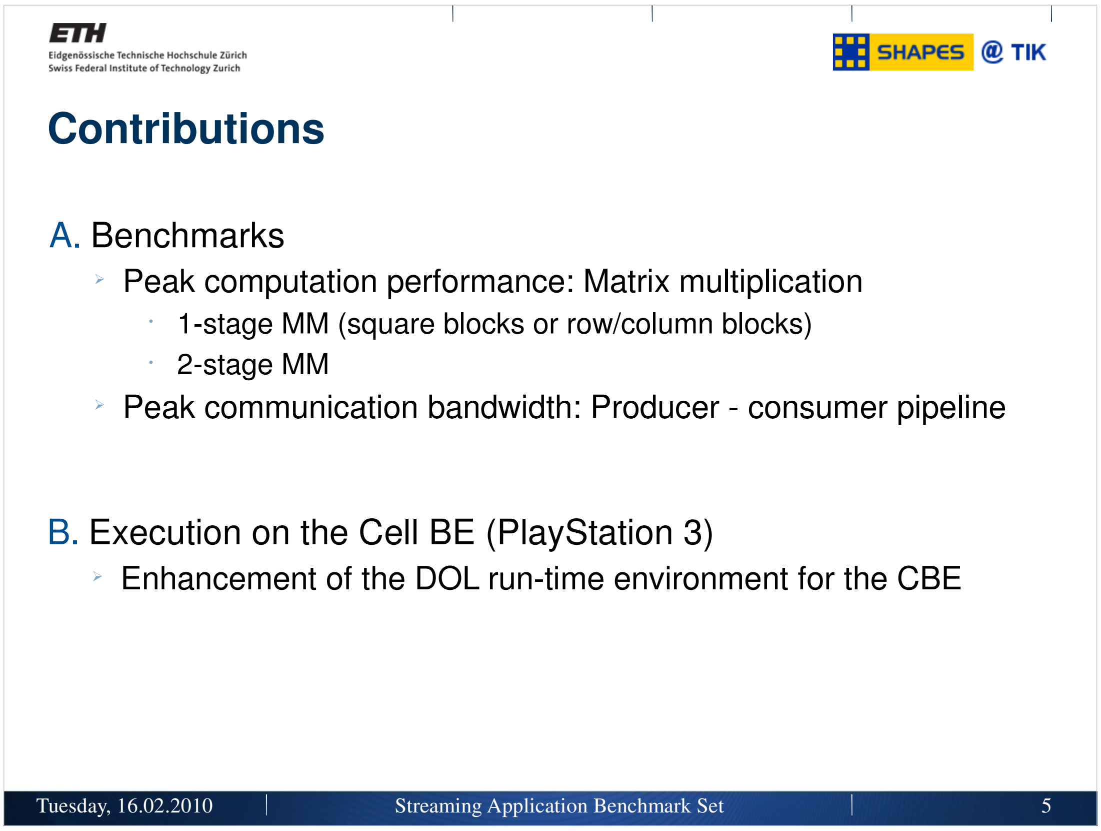


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SHAPES @ TIK

Related Work

- Existing benchmark suites for multiprocessor architectures
  - EEMBC MultiBench, PARSEC, MediaBench II, ALPBench
    → Shared-memory architectures
  - MPIBench, SkaMPI, STREAM
    → Focus on communication overhead of message passing (MPI)
  - StreamIt
    → Benchmarks in StreamIt language



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

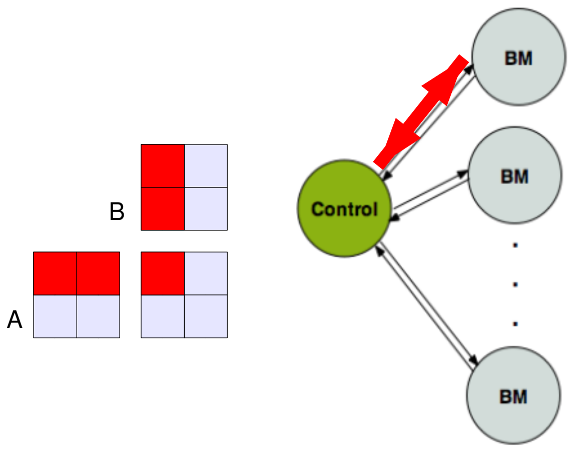
SHAPES @ TIK

Contributions

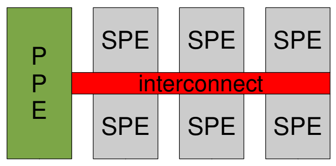
A. Benchmarks
  - Peak computation performance: Matrix multiplication
    - 1-stage MM (square blocks or row/column blocks)
    - 2-stage MM
  - Peak communication bandwidth: Producer - consumer pipeline

B. Execution on the Cell BE (PlayStation 3)
  - Enhancement of the DOL run-time environment for the CBE

1-stage Matrix Multiplication



Cell BE

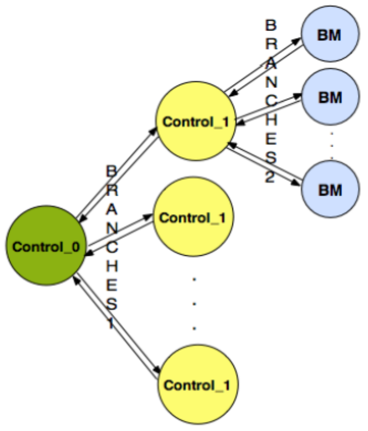


Mapping

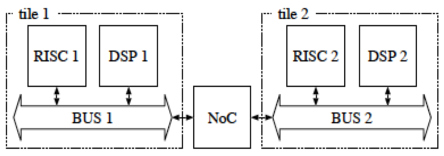
Control to PPE (PowerPC)
Block Multiply to SPEs (DSP)

Control orchestrates MM, i.e. dispatches blocks to corresponding **BM** processes and receives product blocks after sqrt(#blocks) firings

2-stage Matrix Multiplication



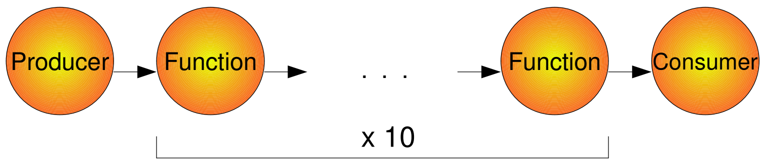
Tiled Architecture (SHAPES)



Exploit in-tile parallelism!

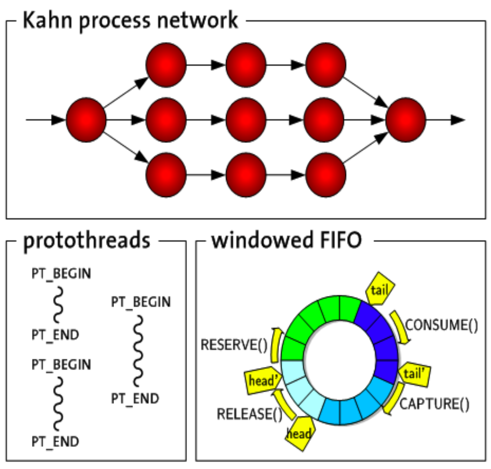
- Alleviate communication bottleneck from/towards Control_0
- Coarse-grained parallelization at 1st stage, finer-grained at 2nd stage
- Control_1 and corresponding BM processes are mapped to the same tile

Producer - Consumer application



- Tokens of maximum size (equal to FIFO capacity) are transferred at each firing

Cell Broadband Engine



- DOL code generation back-end for the efficient execution of KPNs on the Cell BE

- **Protothreads**: multithreading
- **Windowed FIFOs**: inter-process communication
  - Unreliable

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SHAPES @ TIK

DOL run-time environment for the CBE

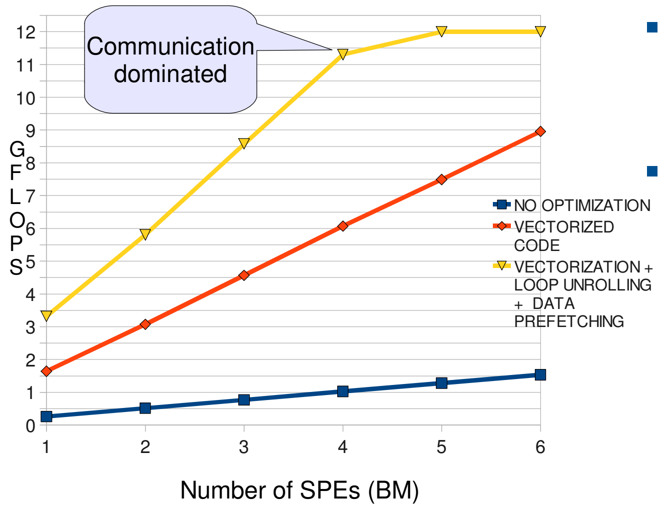
- Enhancement
  - Handling of dynamic memory allocation failures
  - Dynamically chosen size of communication channels
  - Reduction of memory copies needed by the communication protocol
- Result: Reliable platform for experiments on the CBE
  - Benchmark execution on PlayStation 3 under Yellow Dog Linux 6.1

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SHAPES @ TIK

Case Study: 1-stage Matrix Multiplication

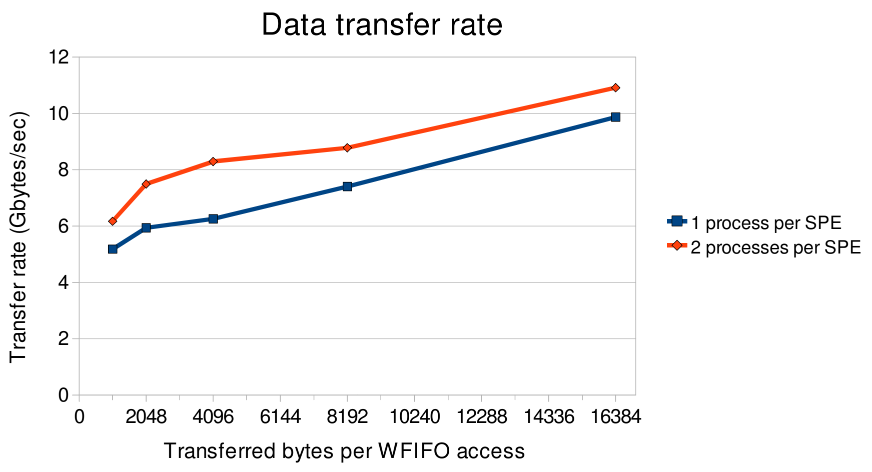
Giga floating-point operations per second (GFLOPS)



Matrix Size: 2048x2048, Block Size: 64x64

- Peak Performance:
  12 GFLOPS
- Cell Superscalar run-time environment
  - MM with matrices / blocks of same size
  - Less efficient implementation
  - Same code optimizations
  - Peak performance: 20 GFLOPS for 5 SPEs
  - → Communication bottleneck!

Case Study: Producer – Consumer



Peak aggregate bandwidth of inter-SPE communication: **10.91 GB/s**

- Maximum token size: 16KB, 1M iterations

Conclusion

- Fully parametrized implementation of a communication-intensive and a computation-intensive benchmark for heterogeneous MPSoC
- Reliable run-time environment for the Cell BE allowing for efficient and scalable execution of parallel applications
  - Need to reconsider WFIFO communication protocol...

Thank you!

References

[1] P. S. Paolucci, "SHAPES Project Web Site." `http://www.shapes-p.org`.

[2] T. Sporer, M. Beckinger, A. Franck, I. Bacivarov, W. Haid, K. Huang, L. Thiele, P. S. Paolucci, P. Bazzana, P. Vicini, J.Ceng, S. Kraemer, and R. Leupers, "SHAPES - A Scalable Parallel HW/SW Architecture Applied to Wave Field Synthesis," in *Proc. Int'l Audio Engineering Society (AES) Conference*, (Hillerod, Denmark), pp. 175–187, Sept. 2007.

[3] P. S. Paolucci, A. A. Jerraya, R. Leupers, L. Thiele, and P. Vicini, "SHAPES: A Tiled Scalable Software Hardware Architecture Platform for Embedded Systems," in *Proc. Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, (Seoul, Korea), pp. 167–172, Oct. 2006.

[4] Lars Schor, "Execution of Process Networks on the Cell Broadband Engine," semester thesis, Department of Information Technology and Electrical Engineering, ETH Zurich, 2009.

[5] Simon Mall, "MPEG-2 Decoder for SHAPES DOL," diploma thesis, Department of Information Technology and Electrical Engineering, ETH Zurich, 2007.

[6] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping Applications to Tiled Multiprocessor Embedded Systems," in *Proc. Int'l Conference on Application of Concurrency to System Design (ACSD)*, (Bratislava, Slovak Republic), pp. 29–40, July 2007.

[7] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. IFIP Congress*, (Stockholm, Sweden), pp. 471–475, Aug. 1974.

[8] I. Bacivarov, W. Haid, K. Huang, and L. Thiele, "SHAPES @ TIK Web Site." `http://www.tik.ee.ethz.ch/~shapes`.

[9] Fabian Hugelshofer, "Scalable Distributing Multi-Linux System for DOL Application," semester thesis, Department of Information Technology and Electrical Engineering, ETH Zurich, 2007.

[10] Embedded Microprocessor Benchmark Consortium, "MultiBench 1.0." `http://www.eembc.org/benchmark/multi_sl.php`.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. Int'l*

Conference on Parallel Architectures and Compilation Techniques (PACT), (Toronto, Canada), pp. 72–81, Oct. 2008.

[12] "Princeton Application Repository for Shared-Memory Computers (PARSEC)." `http://parsec.cs.princeton.edu/`.

[13] MediaBench Consortium, "MediaBench II." `http://euler.slu.edu/~fritts/mediabench/`.

[14] University of Illinois, "ALPBench: All Levels of Parallelism for Multimedia." `http://rsim.cs.illinois.edu/alp/alpbench/`.

[15] Adelaide University, "MPIBench." `http://www.dhpc.adelaide.edu.au/projects/mpibench/`.

[16] "SkaMPI." `http://liinwww.ira.uka.de/~skampi/`.

[17] "PARallel Kernels and BENCHmarks (PARKBENCH)." `http://www.netlib.org/parkbench/html/index.html`.

[18] "The STREAM Benchmark." `http://www.streambench.org/`.

[19] MIT Computer Architecture Group, "StreamIt." `http://groups.csail.mit.edu/cag/streamit/`.

[20] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. Comput.*, vol. C-36, pp. 24–35, Jan. 1987.

[21] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, 2005.

[22] L. Benini, D. Bertozzi, B. Alessandro, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *The Journal of VLSI Signal Processing*, vol. 41, pp. 169–182(14), Sept. 2005.

[23] Barcelona Supercomputing Center, "Cell Superscalar Project." `http://www.bsc.es/plantillaG.php`.

[24] IBM, "Cell Broadband Engine Resource Center." `http://www.ibm.com/developerworks/power/cell/`.

[25] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix Multiplication on Heterogeneous Platforms," *IEEE Trans. on Parallel and Distributed Systems*, vol. 12, pp. 1033–1051, Oct. 2001.

[26] L. Zhuo and V. K. Prasanna, "Optimizing Matrix Multiplication on Heterogeneous Reconfigurable Systems," in *Proc. Int'l Conference on Parallel Computing: Architectures, Algorithms and Applications (ParCo)*, (Aachen, Germany), pp. 561–568, Sep 2007.

[27] Y. Ohtaki, D. Takahashi, T. Boku, and M. Sato, “Parallel Implementation of Strassen’s Matrix Multiplication Algorithm for Heterogeneous Clusters,” in *Proc. Int’l Parallel and Distributed Processing Symposium (IPDPS) - Workshop 1*, (Santa Fe, New Mexico), p. 112a, Apr. 2004.

[28] A. Kalinov, “Scalability Analysis of Matrix-Matrix Multiplication on Heterogeneous Clusters,” in *Proc. Int’l Symposium on Parallel and Distributed Computing / Int’l Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar)*, (Cork, Ireland), pp. 303–309, July 2004.

[29] Mathematics and Computer Science Division, Argonne National Laboratory, “Ian T. Foster’s Online Guide to Designing and Building Parallel Programs.” `http://www.mcs.anl.gov/~itf/dbpp/text/node45.html`.

[30] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele, “Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs,” in *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, (Grenoble, France), Oct. 2009.