Accelerating the Execution of Matrix Languages on the Cell Broadband Engine Architecture

Raymes Khoury, Bernd Burgstaller, and Bernhard Scholz

Abstract—Matrix languages, including MATLAB and Octave, are established standards for applications in science and engineering. They provide interactive programming environments that are easy to use due to their script languages with matrix data types. Current implementations of matrix languages do not fully utilize high-performance, special-purpose chip architectures, such as the IBM PowerXCell processor (Cell). We present a new framework that extends Octave to harvest the computational power of the Cell. With this framework, the programmer is alleviated of the burden of introducing explicit notions of parallelism. Instead, the programmer uses a new matrix data type to execute matrix operations in parallel on the synergistic processing elements (SPEs) of the Cell. We employ lazy evaluation semantics for our new matrix data type to obtain execution traces of matrix operations. Traces are converted to data dependence graphs; operations in the data dependence graph are lowered (split into submatrices), scheduled and executed on the SPEs. Thereby, we exploit 1) data parallelism, 2) instruction level parallelism, 3) pipeline parallelism, and 4) task parallelism of matrix language programs. We conducted extensive experiments to show the validity of our approach. Our Cell-based implementation achieves speedups of up to a factor of 12 over code run on recent Intel Core2 Quad processors.

Index Terms—Programming languages, lazy evaluation, scheduling, data partitioning, math script languages, Cell Broadband Engine architecture.

1 Introduction

Matrix languages, including MATLAB [1] and Octave [2], are established standards for rapid prototyping in scientific and engineering domains. One of the main reasons for the widespread adoption of these languages is their ease of use. They provide interactive execution of code and simple, high-level syntax for matrix calculations. Complex scientific and engineering problems are solved with few lines of code (or even with simple drag and drop graphical user-interfaces) because there exists a cornucopia of commercial and open-source libraries for standard mathematical problems.

Despite their ease of use, matrix languages traditionally have sequential execution semantics and utilize a single thread of execution only. While the performance growth of single-core processors is reaching its limits, scientists and engineers have increasingly encountered large data sets which must be processed efficiently. Thus, the use of matrix languages will plateau in the near future if not adapted to modern parallel computer architectures.

With the advent of hardware accelerators for high-performance computing, such as General Purpose Graphics Processors (GPGPUs) and the Cell Broadband Engine, significant performance boosts over single-core architectures are possible. However, harnessing their computational power is challenging in the context of matrix languages. Hardware accelerators for high-performance computing are attributed to have nonuniform memory accesses and complex parallel programming patterns. Extending matrix languages to execute on high-performance, accelerator architectures can be achieved by adopting either 1) an explicitly parallel programming model which requires users to introduce an explicit notion of parallelism in their matrix language program or 2) an implicitly parallel programming model in which parallelism is elicited from a matrix language program without user intervention. The explicit programming model contradicts the initial design goals of matrix languages, as the users of these languages are most often untrained in concurrent programming. Hence, it is of paramount importance to the continued success of matrix languages that an implicitly parallel model is adopted, which is capable of fully utilizing the computational power of modern accelerated architectures for high-performance computing.

A large body of research [3], [4], [5], [6], [7], [8], [9], [10] already exists on how matrix languages can be parallelized for distributed parallel architectures that were popular before the turn of last century. Some of the parallel extensions developed are reported to offer good performance; however, this performance gain was often paid at the expense of ease of use by the programmer [3]. Little research has been conducted in how matrix languages can be parallelized for modern accelerated architectures that present very different challenges in achieving good performance. Current state-of-the-art techniques (e.g., those employed in MATLAB) for parallelization on multicore CPUs involve using parallel math libraries [11] to exploit data parallelism within matrix operations. Several emerging projects [12], [13] have investigated simple bindings to execute MATLAB functions on GPGPUs, again exploiting the data parallelism of matrix operations. However, these projects offer a naïve approach, neglecting other types of parallelism that exist in matrix

- *R. Khoury and B. Scholz are with the School of Information Technologies, Building J12, The University of Sydney, Sydney, NSW 2006, Australia. E-mail: raymes.khoury@gmail.com, scholz@it.usyd.edu.au.*
- *B. Burgstaller is with the Department of Computer Science, Yonsei University, 134 Sinchon-dong, Seodaemun-gu, Seoul 120-749, Korea. E-mail: bburg@cs.yonsei.ac.kr.*

Manuscript received 6 Oct. 2009; revised 12 Feb. 2010; accepted 1 Mar. 2010; published online 29 Mar. 2010.
Recommended for acceptance by D.A. Bader, D. Kaeli, and V. Kindratenko.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-2009-10-0489.
Digital Object Identifier no. 10.1109/TPDS.2010.58.

language programs, thus resulting in underutilization of their target architectures.

In this paper, we introduce a new framework for the automatic parallelization of matrix languages targeted toward modern hardware accelerators for high-performance computing. We have implemented this framework as an extension to the Octave interpreter running on the Cell Broadband Engine. The Cell Broadband Engine is a heterogeneous multicore architecture which is deployed in the IBM Roadrunner supercomputer [14] as well as in the Sony Playstation 3 computer game console. The Cell consists of a PowerPC core that is connected to several Synergistic Processing Elements (SPEs) via a high-speed interconnect double ring bus. The PowerPC unit is an in-order RISC processor with two hyperthreads, whereas the SPEs are small-sized vector (i.e., SIMD) machines with 256 kB of local memory that is shared for data and instructions. Each SPE delivers approximately 25 GFLOPS peak performance for fused multiply-add operations.

GNU Octave is an open-source alternative to MATLAB (which is a commercial product developed by MathWorks). Octave mimics MATLAB's syntax and has been used in our work because MATLAB does not support PowerPC-based systems, like the Cell architecture.

Our system exploits several types of parallelism in an Octave program to obtain high utilization of the Cell processor:

1. **Data parallelism** is exploited by partitioning large matrices into submatrices and executing operations on the submatrices in parallel.
2. **Instruction-level parallelism** is exploited by executing matrix operations of an execution trace in parallel, if there is no data dependency between them.
3. **Pipeline parallelism** is exploited by overlapping communication between cores with computation of matrix operations.
4. **Task parallelism** is exploited by overlapping execution of the Octave interpreter, construction of the schedule, and execution of the matrix operations on the SPEs.

Lazy evaluation is used to generate execution traces of matrix operations by deferring execution until results are required in the Octave program. A key feature of our system is that partitioned matrix operations are scheduled among the parallel units of the Cell processor in a way that satisfies interoperation data dependencies, prior to execution of the trace. Using estimates of the execution time for each operation, operations can be scheduled such that the utilization of parallel units is improved.

The contributions of our paper are as follows:

1. We introduce a framework for the execution of matrix languages on modern parallel architectures. Our framework exploits both data parallelism and instruction-level parallelism of matrix programs. Instruction-level parallelism is achieved through lazy evaluation of matrix operations.
2. We provide a novel efficient technique for partitioning matrix operations to maximize the parallelism available within an execution trace.
3. We develop accurate time models for estimating the execution time of matrix operations through multivariate regression analysis.
4. We introduce a new heuristic scheduling algorithm which takes into account the estimated execution times of operations and the pipelined nature of execution units.
5. We formulate the scheduling problem as an integer linear program and compare the obtained optimal solution with solutions produced by the heuristic scheduler.
6. We perform an extensive evaluation of our system using nine Octave benchmarks on the Cell and on an Intel Core2 Quad processor.

The paper is organized as follows: In Section 2, we survey related work for parallelizing matrix languages. In Section 3, we give an overview of our system and describe each of the major components. In Section 4, we describe a motivating example that illustrates how an Octave program is executed in parallel with our framework. In Section 5, the lowering process is explained, which decomposes operations on large matrices into operations on smaller matrices. In Section 6, we describe the scheduling problem and provide a heuristic algorithm as well as an integer linear programming formulation that yields the optimal solution. In Section 7, we give an overview of the Cell Broadband Engine architecture and we discuss the implementation of our framework on this architecture. In Section 8, we present the experimental results and discuss the observed performance of our system. We summarize our work and draw our conclusions in Section 9.

2 Related Work

There are a variety of extensions for MATLAB designed to utilize parallel computers. The methods of achieving parallelism, the target architecture, and the extent to which the parallelization process is automated vary from extension to extension. Choy and Edelman provide a survey [3] of 27 projects that extend matrix languages with parallel features. The survey classifies the projects in four main categories:

Embarrassingly parallel: Multiple MATLAB processes are running simultaneously. Communication is only involved when a new process is spawned or a process has completed its task. These kinds of parallel extensions for MATLAB are limited to applications that can adopt the embarrassingly parallel programming scheme.

Message passing (MP): Message passing functionality for MATLAB is provided in the style of the Message Passing Interface (MPI, [15]). Complexity varies, from simple wrappers for MPI functions to more complex abstractions. The programmer has to express the parallelism explicitly.

MATLAB compilers: MATLAB scripts are compiled into an executable form, either directly or through the use of an intermediate language, such as Fortran or C. Some of these projects link their executables with parallel math libraries, while others generate code that utilizes MPI.

Backend support: A single MATLAB process serves as a frontend which generates jobs that are submitted to a computation engine and executed in parallel, often using numerical libraries, like ScaLAPACK. In the following, we

use the categorization provided by Choy and Edelman to examine existing parallel MATLAB systems in relation to our work.

2.1 MP and Embarrassingly Parallel Extensions

MatlabMPI [4] is an MPI implementation that spawns several MATLAB processes that communicate via message passing. The MPI Toolbox [5] is another approach to provide pure message passing functionality for MATLAB. To avoid the complexity of explicit message passing communication, pMATLAB [6] provides the user with distributed numerical arrays (or matrices) together with a mechanism to map these data structures to available processors in the style of High-Performance Fortran. Given a data partitioning, the program is automatically parallelized. pMapper [7] automates the data partitioning mechanism of pMATLAB through a heuristic that generates distribution maps at runtime. pMapper was designed for hardware independence: At installation time, pMapper performs an initialization phase, which generates a performance model for the system. Although pMapper introduces interesting techniques, it mainly addresses signal processing applications and the benchmark results are largely due to simulations.

MathWorks provides parallel extensions for MATLAB in the form of a commercial Parallel Computing Toolbox [8]. The toolbox performs automated parallelization of linear algebra operations on a single machine. Explicitly parallelized code will execute over both multiple processors/cores on a single machine, as well as over multiple machines in a distributed environment. Although MathWorks found in a survey that "reusability of existing MATLAB code was cited as the most important feature of any parallel computing toolset," they have introduced several new constructs to the language to achieve high levels of parallelism.

The goals of our system are largely unaddressed by message passing approaches to parallel MATLAB. First, existing systems are targeted toward distributed parallel architectures. Although they can achieve parallelism on some modern accelerator architectures for high-performance computing, communication overheads can restrict even moderate utilization of the processing elements. Second, most message passing systems require a large amount of intervention from the programmer to achieve parallelism.

2.2 Compilers

FALCON [10] is a programming environment that translates MATLAB code to Fortran 90. FALCON conducts static and dynamic program analyzes that are combined with user input to derive the types and shapes of variables. The user is then required to select from a set of optimizations suggested by FALCON. The code generator utilizes information from the analysis phase; it annotates the generated Fortran 90 code with compiler directives that allow automatic parallelization by the Polaris compiler [16]. Despite achieving speedups of up to 1,000 times over the MATLAB interpreter, FALCON focuses mainly on producing high-performance sequential Fortran code. Parallelization of this code is left largely uninvestigated.

FALCON was followed by the MATLAB Just-In-Time (JIT) compiler MaJIC [17]. With JIT compilation, no static analysis of the code is done. Instead, portions of the code are compiled at runtime in order to achieve better performance than that of purely interpreted code. Although MaJIC did not attempt program parallelization in the compilation process, it remains interesting because, as far as the authors are aware, it is the only research project that uses JIT compilation for MATLAB. JIT compilation is desirable for two reasons. First, it allows MATLAB to remain an interpreted (and untyped) language which is important in facilitating rapid prototyping. Furthermore, it means that optimizations can be applied at runtime that may not be apparent at compile time. In our work, we adopt JIT compilation techniques. In our case, these are not used to generate machine code, but instead allow us to efficiently schedule matrix operations on the Cell architecture.

Otter [18] is a MATLAB compiler which translates MATLAB scripts to parallelized C-code. Otter's C-code is based on MPI and parallel numerical libraries. MATCH [19] is a MATLAB compiler that is targeted toward distributed heterogeneous parallel architectures.

2.3 Backend Support

Jacket [20] is a commercial MATLAB backend for GPGPUs. It is one of several systems that have emerged recently for acceleration of MATLAB on GPGPUs [13], [21]. In Jacket, the programmer casts matrices into GPU matrices, which are transferred to GPU memory. Operations on GPU matrices are executed on the GPGPU by compiling code on-the-fly with the NVIDIA/CUDA infrastructure. Despite theses abstractions, Jacket still requires knowledge of the underlying GPGPU architecture to obtain efficient Jacket/MATLAB code.

Star-P [9] is a commercial product that targets distributed computing environments. The main purpose of the MATLAB client is to distribute tasks to a cluster running the Star-P server, which will run computations and deliver results. Star-P utilizes existing math libraries to perform parallel computations on each server.

3 System Overview

Our framework is a system extension for Octave in the form of a shared library that is loaded by the Octave interpreter at runtime. The framework automatically parallelizes matrix instructions for the Cell Broadband Engine and requires only minimal changes to existing Octave code in order to be used. These changes consist of casting all matrix declarations to a new Octave data type[1] called `p_matrix`. Standard operators, such as $+$, $-$, and $*$, as well as built-in functions, such as $\sin$ and $\cos$ have been overloaded to operate on the `p_matrix` data type.

The underlying idea of our system is to execute several matrix instructions at the same time to optimally harness the computational power of the Cell Broadband Engine. However, the sequential execution semantics of matrix languages do not provide the notion of concurrent execution of matrix instructions, beside splitting the eagerly

1. With a minor change to the Octave System, Octave could use our new parallel matrix data type as the default matrix data type and no manual intervention would be required to execute Octave/MATLAB programs in parallel.

executed matrix instruction into suboperations and distributing them among the parallel processing elements. To further increase the parallelism in Octave programs, we employ lazy evaluation of matrix instructions. Lazy evaluation delays the execution of matrix instructions until the result of an instruction is required. This concept is heavily used in functional programming languages and has numerous applications there, including avoiding unnecessary computations and error conditions, being able to operate on infinite data structures, and defining control flow structures in the language itself [22].

Our framework uses lazy evaluation to collect a *trace* of matrix instructions. The overloaded functions of the new data type facilitate the construction of the trace, which is then analyzed to determine the data dependencies between operations. The data dependencies in the trace loosen the strict sequential ordering of instructions to a partial ordering that allows independent matrix instructions to be executed in parallel. For the trace, a *data dependency graph* $G(I, E)$ is constructed where I is the set of nodes in the graph representing instructions in the trace, and E is the set of data dependencies between pairs of matrix instructions. For example, the lazily evaluated statement `A = B*C` imposes two directed edges (B, A) and (C, A) because the result A of the matrix multiplication depends on the matrix operands B and C. Our framework constructs the data-dependency graph on the fly when matrix instructions are lazily evaluated. Note that the constructed graph is acyclic even for loops. A matrix instruction that is executed multiple times inside a loop is represented by a set of nodes in the graph. For each execution instance of the matrix instruction, there exists exactly one node in the graph.

A trace will continue to grow in length as the program is executed until either 1) an operation is reached that requires the result of an unexecuted instruction in the trace and cannot be lazily evaluated, e.g., displaying the value of a matrix or 2) the length of the trace has reached a certain threshold, i.e., it becomes opportunistic to execute the matrix operations in parallel. If either of these criteria are met, execution of the data dependence graph is triggered.

The first step in execution of the graph is *lowering* the graph. The memory of the SPEs on the Cell architecture is at a premium, i.e., code and data share the same memory which is limited to 256 kB. To be able to compute larger matrices, the framework decomposes matrix instructions into matrix instructions that operate on submatrices. This decomposition of the instructions not only enables the execution of matrix instructions on the parallel processing elements of the Cell, but also exposes data parallelism in the matrix instructions. We refer to this process of decomposing the instructions of the data dependency graph into instructions that operate on submatrices as *lowering* (see Section 5). The decomposition rewrites the original data dependency graph into a lowered data dependency graph, which has an increased number of nodes and dependencies.

The lowered data dependency graph is then scheduled among the parallel processing elements in the underlying architecture (see Section 6). The schedule consists of *instruction streams* for each parallel processing element. The instruction streams are sequences of matrix instructions from the lowered data dependency graph, and are executed on the parallel processing elements in parallel. Synchronization between parallel processing elements is required if an operand of a matrix instruction is not available yet. The mapping of the lowered data dependency graph to instruction streams is performed such that the data dependencies between matrix instructions are satisfied and the total time of execution (makespan) is minimized. Since the scheduling of operations happens at runtime, it is also important that a schedule is produced quickly.

In the final step, the lowered matrix instructions in the instruction streams are executed on the parallel processing elements. This component of the system is referred to as the *computation engine*. The computation engine is abstracted from the details of the underlying architecture and instead viewed only as an asynchronous pipelined MIMD architecture with a shared memory. The architecture executes matrix instructions concurrently in an asynchronous fashion. To hide the communication between memory and the processing element, the architecture utilizes a pipeline. The pipeline stages of a single matrix instruction are assumed to be timely interleaved, and we employ the following pipeline stages in our computation engine:

1. *Data Fetch (df)*: The operands of the matrix instruction are loaded from main memory into the memory of the parallel processing element.
2. *Execute (ex)*: The matrix instruction is executed on the parallel processing element.
3. *Write Back (wb)*: The result of the matrix instruction is written back to main memory.

In contrast to a superpipelined superscalar CPU [23], the matrix instructions are not assumed to be synchronized, i.e., there is no global clock that triggers a new step with a constant period.

This abstraction from the details of the underlying parallel architecture allows the framework to be easily ported to many different architectures by customizing the computation engine. In this work, we implement a computation engine for the Cell Broadband Engine architecture (see Section 7). Each of the SPEs in the Cell processor acts as a processing element and executes a stream of matrix instructions. We call the program that runs on the SPEs a *Matrix Execution Unit* (MEU). The MEUs need to be synchronized globally. An *execution control* mechanism guarantees that a matrix instruction on an MEU is only executed if the operands are already available in main memory. The execution control is run on the PowerPC unit of the Cell.

The software components of our framework are depicted in Fig. 1. The first component is a data-type extension `p_matrix` of the Octave interpreter. The operations of the new data type are overloaded to perform lazy evaluation and to obtain the trace, and consequently, the data dependency graph of the lazy-evaluated matrix instructions. The data dependency graph is passed onto the lowerer that decomposes the matrix instruction into instructions that operate on submatrices. The scheduler computes the instruction streams for the computation engine. The instruction streams are executed on the computation engine. All four components can be executed in parallel, i.e., the Octave interpreter, the lowerer, the scheduler, and the computation engine are executed in separate execution threads, thus allowing an overlapped execution of all four components.

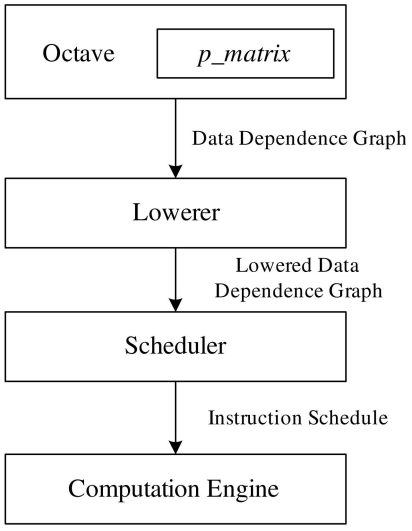


Fig. 1. Software components.

4 Motivating Example

Assume that we want to compute the value of $B = A^i$ using an Octave script, where A is a matrix of dimensions $n \times n$ and i is a positive integer. For the purpose of this example, we let A be a random 100×100 matrix and $i = 3$. A naïve implementation in Octave for calculation of matrix B is given in Fig. 2. After computing the value of B, the result of matrix B is displayed on the screen.

To use our system, the Octave programmer converts a matrix declaration to our custom Octave `p_matrix` data type by wrapping the matrix declaration with the `p_matrix` function. The modified code that uses our system is illustrated in Fig. 3. Note that a user does not have to decide which matrices should be converted to this new data type (all matrices can be safely converted), and the need for this additional data type could be completely eliminated by a simple modification of the Octave interpreter.

When this script is run in Octave, the Octave interpreter will begin executing each statement as follows: Scalar statements are eagerly executed, whereas matrix operations involving the new data-type `p_matrix` are lazily evaluated. The matrix instructions of type `p_matrix` are in lines 1, 4, and 6 of Fig. 3. Because the value of matrix B is not required inside the loop, the system is not forced to execute the trace inside the loop. However, the execution of the

```
1 A = rand(100);
2 i = 3;
3
4 B = A;
5 for k = 1:i-1
6     B = B * A;
7 end
8
9 disp(B);
```

Fig. 2. Octave script for the computation of $B = A^i$.

```
1 A = p_matrix(rand(100));
2 i = 3;
3
4 B = A;
5 for k = 1:i-1
6     B = B * A;
7 end
8
9 disp(B);
```

Fig. 3. Parallel data-type modifications for $B = A^i$.

trace is forced when statement **disp**(B) is reached in line 9 of Fig. 3 to print the value of B.

Traces are kept as internal data structures and a data dependence graph is constructed on the fly for each trace. The data dependence graph shows which operations depend on the results of other operations and determines a partial order in which operations must be executed to yield a correct result. This partial order enables the parallel execution of matrix instructions in contrast to a sequential order of the standard semantics of matrix languages.

Step 1: The declaration of matrix A in line 1 is executed. The conversion of the random matrix to type `p_matrix` adds a constant matrix to the data dependence graph (see Fig. 5a).

Step 2: Matrix A is assigned to variable B in line 4. Again, a constant matrix is added to the data dependence graph (see Fig. 5b), and we denote the first instance by B_0. Note that a deep copy of matrix A is not made on assignment to B_0. For the sake of simplicity, we represent them separately in the data dependence graph.

Step 3: B_0 is multiplied by A and the result is stored in B_1. The result of this operation is not yet required in the program. Hence, execution is deferred and a multiplication operation is added to the data dependence graph (see Fig. 5c). The operation depends on two values—matrix A and matrix B_0. Arcs (A, B_1) and (B_0, B_1) denote these dependencies in the dependency graph.

Step 4: B_1 is multiplied by A and the result is stored in B_2. Again, the result of the multiplication is not required immediately, so another multiplication operation is added to the data dependence graph (see Fig. 5d). The operation depends on two values—matrix A and the result of the previous matrix multiplication operation B_1. Arcs (A, B_2) and (B_1, B_2) denote these dependencies.

In line 9 of our running example from Fig. 3, the value of B_2 is requested to be printed by the **disp**(B) function. However, the value of B_2 has not yet been computed. This causes execution of the Octave program to be halted, while the data dependence graph from Fig. 5d is executed to obtain the result. Note that, in this example, the execution of the trace was forced by a required value. The other cause of executing a trace is that the length of the trace becomes too large. If this is the case, then the execution of matrix instructions and the execution of the Octave interpreter is overlapped.

After constructing the data dependence graph by lazy evaluation, the lowerer partitions matrix operations on large matrices into operations on submatrices. Lowering is

$$B_0A = \left[\begin{array}{ccc|ccc} (b_0)_{1,1} & \cdots & (b_0)_{1,50} & (b_0)_{1,51} & \cdots & (b_0)_{1,100} \\ \vdots & \ddots & & & & \vdots \\ (b_0)_{50,1} & & & & & (b_0)_{50,100} \\ \hline (b_0)_{51,1} & & & & & (b_0)_{51,100} \\ \vdots & & & & \ddots & \vdots \\ (b_0)_{100,1} & \cdots & (b_0)_{100,50} & (b_0)_{100,51} & \cdots & (b_0)_{100,100} \end{array}\right] \left[\begin{array}{ccc|ccc} a_{1,1} & \cdots & a_{1,50} & a_{1,51} & \cdots & a_{1,100} \\ \vdots & \ddots & & & & \vdots \\ a_{50,1} & & & & & a_{50,100} \\ \hline a_{51,1} & & & & & a_{51,100} \\ \vdots & & & & \ddots & \vdots \\ a_{100,1} & \cdots & a_{100,50} & a_{100,51} & \cdots & a_{100,100} \end{array}\right]$$

$$= \begin{bmatrix} (B_0)_{1,1} & (B_0)_{1,2} \\ (B_0)_{2,1} & (B_0)_{2,2} \end{bmatrix} \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} = \begin{bmatrix} (B_0)_{1,1}A_{1,1} + (B_0)_{1,2}A_{2,1} & (B_0)_{1,1}A_{1,2} + (B_0)_{1,2}A_{2,2} \\ (B_0)_{2,1}A_{1,1} + (B_0)_{2,2}A_{2,1} & (B_0)_{2,1}A_{1,2} + (B_0)_{2,2}A_{2,2} \end{bmatrix}$$

Fig. 4. Lowering of operation $B_1 = B_0A$.

necessary because some matrices may be too large to fit into the memory of parallel execution units (e.g., with the Cell architecture), but it has the beneficial side effect of exposing data parallelism in matrix operations. Assume that parallel units have enough memory to store operands of dimensions 50×50; however, our matrices are of dimensions 100×100. We can partition our 100×100 matrices into four 50×50 blocks, and use block matrix multiplication to perform the multiplication operations in our program. Block matrix multiplications work in the same way as regular matrix multiplication, except that instead of multiplying and adding the scalar elements of the two operands, we multiply and add the partitioned submatrices of the operands. Each block of the result matrix is computed using two matrix multiplications and one matrix addition. Fig. 4 shows how this partitioning occurs for the first multiplication operation $B_1 = B_0A$ in our motivating example.

Note that a single matrix operation in our original dependence graph will result in many lowered operations after partitioning. These operations form a new data dependence graph, called the lowered data dependence graph. The lowered data dependence graph for our example is shown in Fig. 5e. Data dependence graph nodes that represent matrix addition and multiplication operations are indicated by $+$ and $\times$ symbols. Each of the operations from the original data dependence graph in Fig. 5d corresponds to several operations in the lowered graph. The lowered data dependence graph exhibits increased opportunities for parallelism.

Once lowering is complete, operations are scheduled among the execution units of the underlying architecture. The aim of the scheduler is to minimize the total execution time (i.e., makespan) by assigning operations to processors and ensuring that data dependencies between operations are not violated: an operation cannot commence before the results of all its operands are available.

Without specific reference to the details of the underlying architecture, it is viewed as a pipelined architecture in which each processor can overlap computation of a matrix operation with data transfers to and from main memory. To produce an effective schedule, it is necessary to estimate the time that each operation will take in each pipeline stage.

A heuristic algorithm uses these execution time estimates to schedule the operations among the execution units. The heuristic works by selecting an unscheduled operation whose operands were scheduled the earliest and assigning that operation in the earliest available slot on a processor. A partial schedule of lowered operations from our motivating example is given in Fig. 6. The schedule has been produced by our heuristic scheduler (see Section 6.3) that ensures that all operands of an operation must have completed before

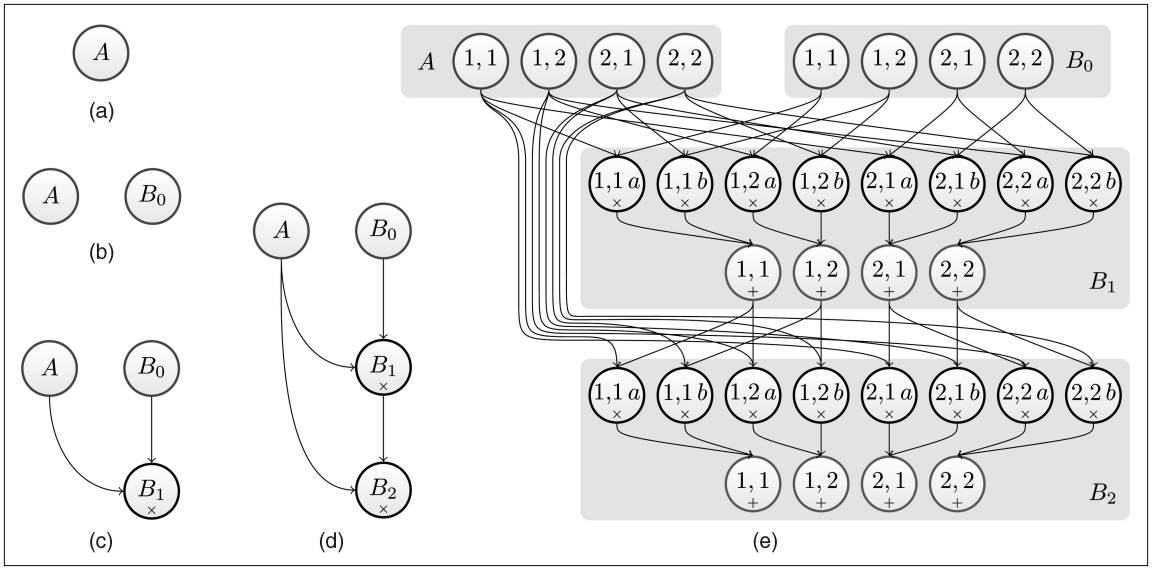


Fig. 5. (a)-(d) Data dependence graph construction. (e) Lowered data dependence graph.

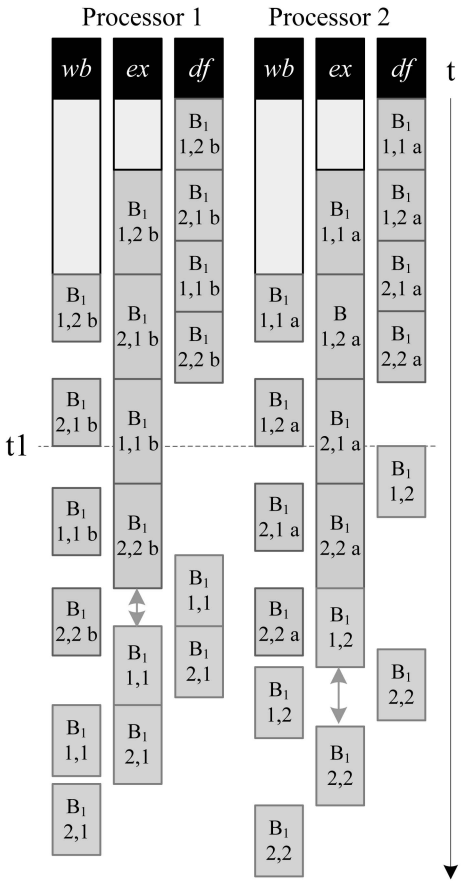


Fig. 6. Schedule of lowered data dependence graph.

the operation can be scheduled. For the schedule, all three stages of the pipeline are considered, i.e., df, ex, and wb. These stages have different durations for different operations, such as multiplication and addition. In Fig. 6, arrows show gaps in the execution, which increase the makespan. Furthermore, label $t1$ marks the point in time when the write back of operation $(B_1)_{1,2}a$ finishes. Operation $(B_1)_{1,2}b$ has already completed at this point. Thus, operation $(B_1)_{1,2}$ that computes the sum of these operations commences.

Once the schedule is produced, the operations are executed on the underlying architecture accordingly. When all operations have completed execution, execution of the Octave program resumes and our sample program completes.

5 Lowering

We identified three requirements to be met by a partitioning scheme: 1) operations should be divided into approximately equal sized portions such that the load on the processing elements is balanced; 2) the number of synchronization points in execution of a trace should be minimized; and 3) the small memory available on the parallel units should be maximally utilized. The third criteria is specific to architectures, like the Cell processor whose SPEs have a small 256 kB local store. Other architectures whose parallel units have access to a large amount of memory have a different concern, which is to determine the size of partitioned blocks in order to obtain best performance. We do not address this concern in our scheme.

With our partitioning scheme, we partition rows and columns of a matrix independently.[2] For a given row (or column) size, we find a uniform partitioning such that the blocks in the partitioning have approximately the same size. The underlying idea is that the number of rows in a partitioned submatrix depends only on the number of rows in the original matrix (and similarly for the columns). This ensures that matrices of compatible dimensions will always have a compatible partitioning, even for matrix multiplication operations. Thereby, we eliminate the need for a complex propagation algorithm for finding partitions in the data dependence graph and the need for artificial barriers between dependent operations, thus increasing the parallelism available in the trace.

To be applicable to a wider range of architectures, our partitioning scheme abstracts from the underlying hardware, but only to the extent that performance is not sacrificed. We were able to reduce the dependencies of our partitioning scheme to two quantities from the underlying Cell BE architecture: First, we need to determine the maximum number of matrix elements S that a matrix can contain. The local store of a Cell SPE provides 256 kB of memory to be shared by program code and data. We use a small (≈26 kB) kernel on each SPE to execute matrix operations. The remaining 230 kB of the local store are available to matrix blocks. Because of Cell-related implementation techniques discussed in Section 7, we can devote up to 38 kB to a single matrix block, which amounts to $S = 9{,}216$ matrix elements per block in single precision or $S = 4{,}624$ matrix elements per block in double precision. Second, we need to determine a divisor δ for the number of rows and columns of a block. For several Cell-related reasons discussed in Section 7, the number of rows and columns in a block must be a multiple of $\delta = 4$ with single precision and a multiple of $\delta = 2$ with double precision.

Initially, we employed least-squares fitting to partition an $n \times m$ matrix into p rows and q columns such that blocks would be close to the ideal, real-valued, block size $\frac{nm}{pq} \approx S$. Because partitioning is a timing-critical runtime operation, we replaced least-squares fitting by the following approximation to determine the number of rows p and columns q of a matrix partition:

$$p = \left\lceil \frac{\lceil \frac{n}{\delta} \rceil}{\lfloor \frac{\sqrt{S}}{\delta} \rfloor} \right\rceil, \quad \text{and } q = \left\lceil \frac{\lceil \frac{m}{\delta} \rceil}{\lfloor \frac{\sqrt{S}}{\delta} \rfloor} \right\rceil.$$

Therein, $\lceil \frac{n}{\delta} \rceil$ is the number of groups-of-δ-rows of a partition, and $\lfloor \frac{\sqrt{S}}{\delta} \rfloor$ is the number of groups-of-δ-rows that fit in a single matrix block. Matrix blocks consist of up to $\delta \lfloor \frac{\sqrt{S}}{\delta} \rfloor$ rows and columns.

The number of rows k_i and columns l_j of a block are given by

$$k_i = \delta\left(\left\lfloor \frac{\lceil \frac{n}{\delta} \rceil}{p} \right\rfloor + r_i \right), \quad \text{and } l_j = \delta\left(\left\lfloor \frac{\lceil \frac{m}{\delta} \rceil}{q} \right\rfloor + s_j \right),$$

where $\sum_{i=1}^{p} r_i = \lceil \frac{n}{\delta} \rceil \bmod p$ with an arbitrary choice for $r_i \in \{0, 1\}$, and $\sum_{j=1}^{q} s_j = \lceil \frac{m}{\delta} \rceil \bmod q$ with an arbitrary choice

2. The design decisions that lead to this scheme are described in the accompanying technical report [24].

for $s_j \in \{0, 1\}$. For consistency across matrices, we set $r_i = 1$ and $s_j = 1$ for the smallest indices i and j of blocks $A_{i,j}$ of a matrix partition.

With our partitioning scheme, the number of rows and columns of a matrix block must be a multiple of δ, which requires padding of up to $\delta - 1$ additional rows and/or columns of zeroes. The number of elements of the resulting $n' \times m'$ matrix is bound by

$$n \times m + (\delta - 1)(n + m) \geq n' \times m'.$$

6 Scheduling of Matrix Operations

The task of the scheduler is to generate the matrix instruction streams for each MEU. The input of the scheduler is the lowered dependence graph annotated with estimated time durations for each pipeline stage of an instruction. The objective of the scheduler is 1) to produce a feasible schedule, i.e., instructions for operands of a matrix instruction must not be scheduled after the instruction, 2) to generate the instruction streams for the MEUs as fast as possible, and 3) to minimize the *makespan*, i.e., the wall-clock time needed to execute the parallel schedule on the MEUs. The second and third objectives are hard to achieve because scheduling is an NP-hard problem [25]. Recognizing the difficulty of finding an optimal solution for the general problem, many heuristics have arisen [26].

Our scheduling problem can be seen as a variant of the offline scheduling problem with task precedence, arbitrary time durations for tasks, and uniform workers/processors. The data dependence graph resembles the task-precedence graph, the workers are the MEUs, and the time durations are estimated ahead of time for each lowered matrix instruction. For this problem, there exists an approximation algorithm [27] that computes a solution in polynomial time and gives an approximation guarantee for the quality of the solution of $2.33 - 1.33m$, where m is the number of edges in the task-precedence graph. The algorithm is based on rounding of a relaxed linear programming solution and is not viable for our framework due to high execution overheads for our problem sizes. Furthermore, the subclass of offline scheduling problems with task precedence do not have the notion of a pipeline for each of the workers leading to imprecision in the pipeline modeling.

In the following, we discuss how to 1) find a time model for the time durations for each pipeline stage of a lowered matrix instruction, 2) model a mathematical program for the pipelined scheduling problem with task precedence, and 3) devise a greedy algorithm for solving our scheduling problem efficiently.

6.1 Time Model

An accurate time model for the scheduling is crucial to find an effective schedule. We compute a time model for each type of matrix instruction in our framework ahead of execution. Multivariate polynomial functions that depend on the number of rows and columns of the input and output matrices are employed for our time model. The coefficients of the multivariate polynomial are computed using profiling and the *Ordinary Least Square* (OLS) method [28].

For scheduling, we require a fairly accurate time model that estimates the durations of the three phases of a matrix instruction before executing it. For computing the time model, we instrument the virtual machines that run on the SPEs. We obtain profiling information as a side-effect of execution. The instrumented version of the virtual machine is only executed for profiling purposes to obtain the execution times for each pipeline stage of an instruction. Furthermore, we have a carefully crafted input program that executes each type of matrix instruction for every possible input problem size several times. Note that it is feasible to profile every input problem size, since the matrices are limited by the SPE's small data and program memory of 256 KB.

The Cell Broadband Engine offers hardware counters [29] for performing the measurements with very high precision. We modified Bryant's performance measurement library [30] for the Cell architecture that provides special intrinsics for reading the hardware cycle counters. The execution time measurements on the SPEs have little variation due to the lack of data caches. However, we observed a higher volatility in the measurements for the data-fetch and write-back stages (see Section 8), which is not surprising, since the ring bus of the Cell that connects SPEs, memory, and the PPE is shared.

The coefficients of the multivariate polynomial are chosen such that the deviation of the polynomial function from the measured time durations is minimized. Assume that we have l time measurements obtained by profiling, which consists of the input size vector $\vec{n}_k$ describing the input problem size of the kth measurement of a type of matrix operation, and the three durations $\Delta_{df}(k)$, $\Delta_{ex}(k)$, and $\Delta_{wb}(k)$ for each phase of the pipeline. For example, a matrix addition has two elements in the input-problem size vector describing the number of rows and columns of the two matrices that are to be added.

We seek a multivariate polynomial for each pipeline phase of each matrix instruction of the form $t_\xi(\vec{n}) = \sum_k a_k g_k(\vec{n})$, where $g_k(\vec{n})$ is a multivariate term of the multivariate polynomial and $a_0, \ldots, a_n$ are the coefficients such that the error $R_\xi = \sum_{k=1}^{l} (t_\xi(\vec{n}_k) - \Delta_\xi(k))^2$ becomes minimal (where ξ is either *df*, *ex*, or *wb*).

In the following table, we give the multivariate polynomials for some of the matrix instructions, the data-fetch, and write-back duration.

$t_\xi(\vec{n})$	Operation
$a_0 + a_1 n_1 + a_2 n_2$	data-fetch and write-back
$a_0 + a_1 n_1 n_2$	scalar instr. execution
$a_0 + a_2 n_1 + a_1 n_1 n_2 n_3$	matrix multipl. execution

The problem sizes n_1, n_2, and n_3 represent the rows and columns of the input matrices. For matrix multiplications, we do not need to specify the number of rows of the second matrix because it is equal to the number of columns of the first matrix. Furthermore, we add an additional term $a_2 n_1$ to account for the overhead of the inner-most loop in the matrix calculation to obtain a better fit of the profile data. We used standard methods to obtain the coefficients [28]. In abuse of notation, we denote with $t_\xi(i)$ the function $t_\xi(\vec{n})$, where $\vec{n}$ is the problem size of instruction $i \in I$.

6.2 Mathematical Program

The mathematical program for finding pipelined schedules is not practical, but for small problem sizes, it gives us a yardstick to compare how good heuristics perform in

comparison to the optimal solution. Because the problem is intractable, we cannot hope for computing schedules with the mathematical program for medium-sized to larger problems.

We develop an integer linear program that computes an optimal schedule for a given problem instance that consists of the set of matrix instructions $I = \{1, \ldots, n\}$, their data dependencies $E \subseteq I \times I$, where $(i, j) \in E$ denotes that j depends on i, and the time parameters $t_{df}(i)$, $t_{ex}(i)$, and $t_{wb}(i)$ for instructions $i \in I$. For the model, we introduce the variables $x_{ij} \in \{0, 1\}$, for all $i, j \in I$, $t_i \in \mathbb{R}^+$, for all $i \in I$, and $z \in \mathbb{R}^+$, where z is the makespan of the schedule, t_i is the start time of an instruction, and x_{ij} are elements of an adjacency matrix for the *schedule graph*, which is a directed rooted graph that forms the instruction streams of the matrix execution units.

We define a schedule graph as a directed graph that has a dedicated start node with at most p outgoing edges, and no incoming edges. The start node is an artificially introduced matrix instruction. The remaining nodes in the schedule graph have exactly one predecessor node and, at most, one successor node. The successor and predecessor node must not be the node itself, i.e., in the graph, we do not allow self-loops. The successor nodes of the start node are the first nodes of a stream that are executed by a matrix execution unit. Their successor nodes are the second instructions of the instruction streams and so on.

We have at most p outgoing edges for the start node, and hence, at most p streams. The structure of a schedule graph is depicted in Fig. 7a. The linear constraints that ensure that elements x_{ij} of the adjacency matrix form a schedule graph are given in Fig. 7b.

For the task-precedence relation E, we introduce the time constraints $t_i + t_{df}(i) + t_{ex}(i) + t_{wb}(i) \leq t_j$, for all $(i, j) \in E$ that forces an instruction $j \in I$ to be scheduled after its operands $i : (i, j) \in E$ are completed. The time t_i is a global time for all matrix execution units, i.e., operands may be scheduled on different matrix execution units and all matrix execution units have the same wall-clock time.

For two subsequent instructions in a stream, we ensure that the duration spans of the three pipeline stages data fetch, execute, and write back do not overlap. For each stage, a time constraint is introduced,

$$\begin{aligned}
\sum_{i=1}^{n} \big(t_i + t_{df}(i)\big) x_{ij} &\leq t_j && j \in I, \\
\sum_{i=1}^{n} \big(t_i + t_{df-ex}(i)\big) x_{ij} &\leq t_j + t_{df}(j) && j \in I, \\
\sum_{i=1}^{n} \big(t_i + t_{df-ex-wb}(i)\big) x_{ij} &\leq t_j + t_{df-ex}(j) && j \in I,
\end{aligned}$$

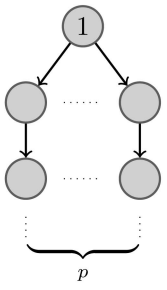
where $t_{df-ex}(i) = t_{df}(i) + t_{ex}(i)$ and $t_{df-ex-wb}(i) = t_{df}(i) + t_{ex}(i) + t_{wb}(i)$, for all $i \in I$. The left-hand sides of the constraints are summations of the time durations multiplied by x_{ij} over all predecessors. However, in a schedule graph, there exists at most one predecessor modeled by the constraints for the schedule graph.

The expansions of the above constraints have a nonlinear term $t_i x_{ij}$, which we linearize by introducing a new variable y_{ij}. This new variable replaces the quadratic term $t_i x_{ij}$, and we add linear constraints to the program that force the equivalence. The linear constraints are developed by using standard techniques [31], i.e., a new variable $y_{ij} \in \mathbb{R}^+$, for all $i, j \in I$ is introduced and following constraints are added:

$$\begin{aligned}
y_{ij} &\leq U x_{ij} && i, j \in I, \\
t_i + U x_{ij} - U \leq y_{ij} &\leq t_i && i, j \in I,
\end{aligned}$$

where U is the sum of all time parameters, which is an upper bound for y_{ji} and t_{ji}.

The makespan is determined by the matrix instruction that terminates last, i.e., $t_i + t_{df-ex-wb}(i) \leq z$, for all $i \in I$, and the objective function of the mathematical program is to minimize the makespan z. The integer linear program is given in the accompanying technical report [24].



(a)

$$\begin{aligned}
&\textstyle\sum_{i=1}^{n} x_{1i} \leq p \\
&x_{i1} = 0 && i \in I \\
&\sum_{j=1}^{n} x_{ij} \leq 1 && i \in I \setminus \{1\} \\
&\sum_{j=1}^{n} x_{ji} = 1 && i \in I \setminus \{1\} \\
&x_{ii} = 0 && i \in I \setminus \{1\}
\end{aligned}$$

(b)

Fig. 7. Schedule graph and schedule graph constraints. (a) Graph. (b) Constraints.

6.3 Greedy Algorithm

Algorithm 1. Heuristic Scheduler

```
 1 for all k ∈ {1, ..., p}
 2   s_df(k) ← 0
 3   s_ex(k) ← 0
 4   s_wb(k) ← 0
 5 for all i ∈ I
 6   c_i ← |d⁻(i)|
 7   e_i ← 0
 8   if |d⁻(i)| = 0
 9     queue i with cost 0
10 while queue not empty
11   dequeue i
12   k ← minarg_r{h(i, r)}
13   t_i ← h(i, k)
14   add i to stream k
15   s_df(k) ← t_i + t_df(i)
16   s_ex(k) ← t_i + t_df-ex(i)
17   s_wb(k) ← t_i + t_df-ex-wb(i)
18   for all j ∈ d⁺(i)
19     c_j ← c_j − 1
20     e_j ← max(e_j, t_i + t_df-ex-wb(i))
21     if c_j = 0
22       enqueue j with cost e_j
```

The integer linear program for scheduling matrix instructions is intractable. To overcome this problem, we devise a simple heuristic, as shown in Algorithm 1, which adopts the ideas of list scheduling for pipelined architectures and exhibits a worst-case runtime of $\mathcal{O}(n \log n + m)$, where n is

the number of instructions and m is the number of dependencies in the task-precedence graph. The sets $d^-(i)$ and $d^+(i)$ denote the predecessor and successor sets of an instruction $i \in I$ in the task-precedence graph.

The underlying idea of the heuristic is to find a topological order for the acyclic data dependency graph $G(I, E)$. This total order $(i_1, \ldots, i_n)$ for the instructions I has the property that for each data dependency $(i, j) \in E$ instruction, j is listed after instruction i in the total order. The topological order ensures that instructions never deadlock in its execution, i.e., the operands are either available or the instructions have to wait a finite amount of time for their operands. The topological order is computed with the counter c_i for an instruction $i \in I$. Initially, the counters are set to the number of incoming edges, i.e., $|d^-(i)|$. Only instructions with counter values of zero can be scheduled and are maintained in a priority queue (min-heap). The priority is determined by the earliest point in time when an instruction can be scheduled. After scheduling an instruction i on one of the streams, the counters of its successors $j \in d^+(i)$ in the task-precedence graph are decremented by one. The successors are scheduled (put into the queue) as soon as their operands have been allocated to a stream.

From the set of instructions whose operands have been scheduled (or do not have operands), we choose the instruction that can be scheduled the earliest on the time line. The earliest scheduling time e_i of an instruction $i \in I$ is determined by the completion time of the instruction's operands. When an instruction is scheduled, the earliest scheduling time of its dependent instructions is updated (cf. line 20 in Algorithm 1).

We have for each instruction exactly one enqueue operation and one dequeue operation. Hence, we have a worst-case complexity for the operations of $\mathcal{O}(n \log n)$. We need to update the earliest scheduling time e_i of the instructions, and hence, have a worst-case complexity of $\mathcal{O}(m)$ for the update operations, resulting in a worst-case runtime complexity of $\mathcal{O}(n \log n + m)$ for Algorithm 1. Note that in the worst-case complexity consideration, we have the underlying assumption that the number of processors p is constant.

Once the earliest instruction is selected, we choose the stream of a matrix execution unit that can execute the instruction at the earliest. For the selection of the stream, we use a slot function $h(i, k)$, which computes the earliest possible start of instruction i in stream k. The slot function takes into account the completion times of the data-fetch, execute, and write-back phase of the previous instruction. We keep track of the completion time with $s_{df}(k)$, $s_{ex}(k)$, and $s_{wb}(k)$ for each phase of stream k. The slot function is given by

$$h(i,k) = \max(\max(\max(e_i, s_{df}(k)) + t_{df}(i), s_{ex}(k)) + t_{ex}(i), s_{wb}) - t_{df-ex}(i),$$

and is depicted in Fig. 8. The three max functions are required to check whether the completion time of the data-fetch stage is before or after the earliest start time of the instruction and/or the execute and write-back stage are limited by the completion time of the previous instruction. In the example, the earliest execution time $e(i)$ of instruction i is

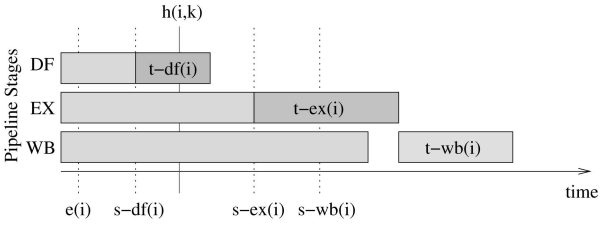


Fig. 8. Example for slot function $h(i, k)$.

before the completion of the data-fetch stage of the previous instruction s_{df}. Thus, the completion of the data-fetch stage of the previous instruction is taken. We add to this point in time the estimated duration of the data-fetch time and compare it with the completion of the execution stage of the previous instruction to compute the earliest point in time for commencing the execution stage of i. In the example, the completion of the execute stage of the previous instruction takes longer time, and hence, determines the starting point for the execution of i. The last max function considers the completion of the write back, and from there, we can subtract the duration of data fetch and execute to obtain the earliest point in time to slot instruction i into stream k.

7 Cell Computation Engine

The Cell Processor is a heterogeneous multicore architecture that was jointly developed by IBM, Sony, and Toshiba [32]. The Cell consists of a 64-bit PowerPC core (PPE), eight SIMD cores called SPEs, a memory-interface controller, and an I/O controller. The PPE and SPEs communicate through a high-speed element interconnect bus (EIB).

The PPE is the Cell's main processor designated to run the operating system, coordinate the SPEs, and perform the control-intensive part of applications. SPEs are designed for high-performance data streaming and for data-intensive computations. Their memory hierarchy consists of a 128 $\times$ 128-bit SIMD register file, 256 kB of local store memory and the off-chip main memory shared with the PPE. SPEs can run SIMD operations at four different granularities: 16-way 8-bit integers, eight-way 16-bit integers, four-way 32-bit integers, four-way single-precision floating-point numbers, or two-way 64-bit double-precision floating-point numbers. The 256 kB local store of an SPE is shared for code and data. Each SPE can only access the code and data in its own local store. DMA transfers are used to move data between local store and main memory and between local stores of different SPEs. DMA transfers are asynchronous and enable SPEs to overlap computation with communication. Unlike caches, SPE local stores must be explicitly managed by software.

At the core of our Cell computation engine are the MEUs that are small virtual machines, which run on individual SPEs and execute matrix operations. There is one MEU on each SPE. Multiple MEUs capable of executing matrix operations concurrently give the system its *superscalar* properties. Each MEU waits to be notified by the PPE that a matrix operation is ready for execution. The PPE uses the Cell's mailbox mechanism to inform an MEU of the memory location of a ready operation.

MEUs use Direct Memory Access (DMA) list commands to transfer the operands involved in the matrix operation from main memory to the local store of the MEU's SPE. Once the transfer is complete, the MEU executes the matrix operation and transfers the result of the operation back to the main memory (employing again DMA list commands). To achieve high performance, the latency of memory transfers between main memory and an SPE's local store must be hidden by overlapping them with computation. With our MEUs, we employ a variant of multibuffering known as triple buffering to hide data-transfer latencies. Three buffers are allocated on the local store of each SPE. At any point in time, exactly one of the following data items will be stored in each of the buffers: First, the operands of the current operation being executed. Second, the result of the current operation being executed. Third, the result of the last operation that was executed as it is transferred out *or* the operands of the next operation to be executed as they are transferred in.

Triple buffering results in *pipelined* execution of operations on each of the MEUs. There are three stages in the execution pipeline: 1) **Data Fetch**: The operands of an operation are loaded from main memory into the memory of the SPE. 2) **Execute**: The operation is executed on the SPE. 3) **Write Back**: The result of the operation is written back to main memory.

The small local stores of SPEs limit the size of matrices that can be operated on by the SPEs. The MEU virtual machine code is 26 kB in size, leaving 230 kB of space for matrices. Triple buffering divides this space further into three buffers of size $\approx$76 kB, each of which should be capable of storing the operands of a single matrix operation. We assume up to two operands for each matrix operation, yielding a maximum size of $\approx$38 kB for each matrix block. The partitioning scheme described in Section 5 results in blocks that are up to $\delta\lfloor\frac{\sqrt{S}}{\delta}\rfloor \times \delta\lfloor\frac{\sqrt{S}}{\delta}\rfloor$ in dimensions, where $\delta = 4$ for single precision and $\delta = 2$ for double precision. This results in an effective block size for each operand of $S = 9{,}216$ scalar elements for single precision and $S = 4{,}624$ scalar elements for double precision.[3]

The values chosen for the divisor, $\delta = 4$ for single precision and $\delta = 2$ for double precision, ensure that the length of rows in a block are always a multiple of 16 bytes. This is necessary because each row of the matrix block is transferred by a separate DMA command, and DMA commands on the Cell are limited to sizes that are multiples of 16 bytes. Block dimensions, which are a multiple of 16 bytes, are also necessary to utilize the SIMD operations on the SPEs that operate on 128-bit (16 byte) vectors. Matrix libraries, such as the Large Matrix Library [33] and the SPE BLAS library [33] provided for the SPEs require matrices to be of these dimensions.

Optimal DMA performance is seen when transfer sizes are multiples of 128 bytes [32]. In single precision, the maximum length of a row in a block is 384 bytes with our partitioning scheme. Thus, blocks that reach this maximum length will achieve optimal transfer rates. By choosing $\delta = 32$ for single precision and $\delta = 16$ for double precision, block row lengths are *always* multiples of 128 bytes. However, this would result in additional space overhead in the padding of matrices. Similar considerations are due for the alignment of matrix blocks in memory, for which we refer again to the accompanying technical report [24].

A fast and efficient execution control mechanism is required to deliver scheduled operations from the PPE to the SPEs and track the completion of operations. On the PPE, each SPE is modeled as an *execution queue*. Unexecuted operations can be enqueued onto the end of the queue and operations that have finished execution can be dequeued from the front. At any point in time, the execution queue contains several kinds of operations: Unexecuted operations are operations that the PPE has offered to the SPE, but which the SPE has not yet acknowledged. Executing operations are operations that the SPE has acknowledged and either started data transfer or execution of. Executed operations are operations which the SPE has finished executing and has returned the result to main memory, but which the PPE has not yet acknowledged as complete by dequeuing them from the execution queue.

The PPE offers an unexecuted operation to an SPE by placing the memory address of the operation in the mailbox of the SPE. The SPE acknowledges reception of the operation by removing it from the mailbox. Because SPE mailboxes are restricted to four mails, our execution queues are restricted to four unexecuted operations.

To enqueue an unexecuted operation OP_u onto an execution queue, all of OP_u's operands must be available in main memory. If OP_u depends on the results of other operations, these must be computed by an SPE and returned to main memory prior to OP_u being added to the execution queue. The scheduler should produce a schedule that ensures this requirement. However, indeterminisms in the architecture can cause variations in the actual execution times of operations, which cannot be anticipated by the scheduler. To ensure that all operands of an operation have been executed, we introduce a *guard count* value to each operation. The guard count is an integer value that represents the number of operands of an operation, which have not yet been computed. Before the execution of any operation, every operation's guard count is equal to the total number of operands of the operation. When an executed operation OP_e has been acknowledged by the PPE as being complete, the guard count of all the operations, which depend on OP_e, are decremented by 1. This means that it is safe to place an unexecuted operation on the execution queue when its guard count is zero.

To signal the completion of an operation, the SPE uses DMA commands to write a counter value to a predetermined location in main memory. The PPE polls this memory location and when it changes knows that an operation is complete and it is safe to dequeue the instruction from the execution queue. This interprocessor communication technique is chosen over other techniques for performance reasons.

The execution control process consists of the PPE examining each execution queue in turn. It enqueues as many unexecuted operations as possible onto the execution queue, in the order that they are specified in the processors' schedule. The SPE is notified of these operations through the mailbox.

3. Assuming 4 bytes for single precision numbers and 8 bytes for double precision.

The PPE then reads the counter value and dequeues any complete operations from the execution queue, decrementing the guard count of operations that depend on the completed operation. This process continues until all operations in the current schedule have completed execution.

8 Experimental Results

In the experimental evaluation of our framework, we focused on the performance improvements that users can expect from running Octave application code with our framework on the IBM Cell processor. We investigated two likely scenarios of use: First, a user currently using *Octave* on a contemporary Intel multicore CPU runs the application code on a Cell with our framework. Second, a user currently using *MATLAB* on a contemporary Intel multicore CPU runs the application code on the Cell, using our framework. We investigated our approach with respect to scalability to larger numbers of SPEs, accuracy of our time model for scheduling matrix computations on the Cell, quality of schedules derived by our heuristic scheduling algorithm, and the extent to which our framework is able to utilize the Cell SPEs.

Our experimental setup consisted of an IBM BladeCenter QS22 running Red Hat EL Server 5.4 on Linux kernel version 2.6.18. We used the IBM Cell SDK 3.1.0 and Octave 3.0.3. Our framework was compiled with GCC 4.1.1 under the -O3 optimization flag. For our experiments, we used only one out of the two Cell processors available on the BladeCenter. This restriction to eight SPEs was not caused by a limitation of our framework, but rather by nonuniform-memory-access (NUMA) issues between different Cell processors of the BladeCenter QS22. Such NUMA issues will be faced by QS22 software in general, and are therefore not confined to our framework. We are investigating these issues, but we felt they are outside the scope of this paper. To prevent NUMA issues from perturbing the measurements, we configured our framework to utilize eight SPEs of one QS22 Cell processor only. We used the following benchmarks to evaluate the performance of our system:

- **dft**: Computation of the discrete Fourier transform (DFT) of a series of signals.
- **dma**: A synthetic benchmark with many small, independent matrix multiplications.
- **hill**: Encryption and decryption using Hill ciphers.
- **hits**: Computation of the Hyperlink-Induced Topic Search (HITS) algorithm for estimating the importance of a Web page.
- **kmeans**: Computation of the k-means clustering of a 2D point set.
- **leontief**: Computation of a Leontief input-output model, used to predict performance of economies.
- **markov**: Computation of a Markov chain.
- **neural**: Training of a single-layer neural network.
- **reachability**: Computation of the reachability matrix of a graph.

For the first scenario, we compared our framework on the Cell with the execution times achieved on a default Octave installation on an Intel Core2Quad Q9550 2.83 GHz.

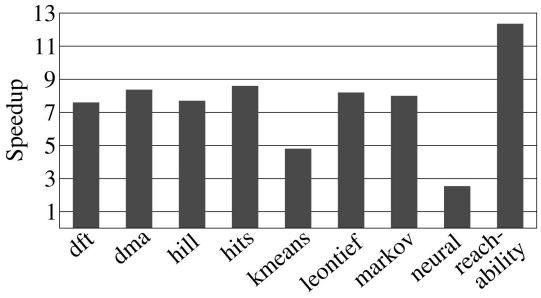


Fig. 9. Improvement of Cell Octave matrix engine over Octave on IA-32.

The standard Octave installation on Intel uses single-threaded ATLAS [11] BLAS libraries, which utilize the SSE3 extensions of the processor. As depicted in Fig. 9, the matrix engine of our framework achieves speedups of up to a factor of 12 over standard Octave on Intel. For most benchmarks, we achieve speedups of over 7. The k-means clustering ("kmeans") and neural network ("neural") benchmarks achieve lower speedups. We attribute those to benchmark operations that we have not yet implemented on the SPEs, which caused these operations to be executed on the PPE. Another reason for the lower speedups with these two benchmarks is that the time models for operations in those benchmarks require further fine tuning (see Section 6.1).

For the second scenario, we have run our benchmark suite on the latest version of MATLAB (7.8) that utilized all four cores of the before-mentioned Intel CPU. The Cell Matrix Engine achieved speedups of up to $8\times$ in comparison with the Intel/MATLAB combination. When Octave is executed on the Cell PowerPC without utilizing the SPEs, our framework achieved speedups of several hundred times.

Fig. 10 shows the scalability of our framework in terms of the number of SPEs. Performance deviates from linear speedup depicted by the dotted line. Lowering and scheduling of operations constitute a portion of the computation that is currently not parallelizable for a single trace, which explains the supra-linear speedup.

To obtain effective schedules from the scheduler, it is important that the estimated execution times of operations are accurate. To verify this, we computed the makespans estimated by the time model and the greedy heuristic, and compared them to the observed execution makespans determined with the help of hardware performance counters on the Cell processor. A histogram of the deviations between the two makespans is shown in Fig. 11. The majority of deviations are close to 0 percent, indicating an accurate time model. The median of the deviations is 1.3 percent and the skewness is -0.54, which means that the the distribution is slightly skewed toward the right. The skewness of the distribution is caused by the overhead of the execution control, which synchronizes dependent matrix instructions on different streams. This execution overhead is not taken into account for the estimated makespan of the heuristic and underestimates the makespan accordingly. Note that Fig. 11 includes deviations for more than 100 data dependence graphs generated by all benchmarks except for benchmarks "neural" and "kmeans." We excluded these benchmarks because they have a high portion of matrix-vector multiplications that are currently

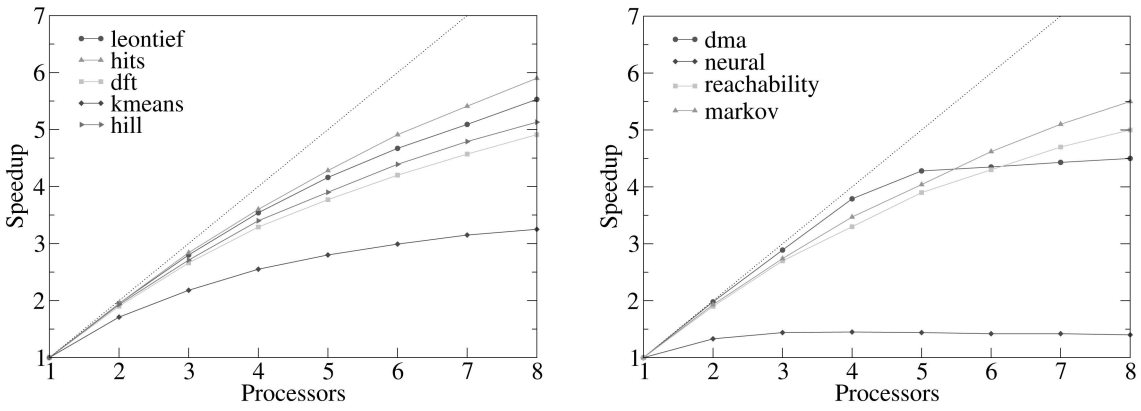


Fig. 10. Performance scalability of the benchmark suite on the IBM BladeCenter QS22.

dealt within the system as matrix-matrix multiplications. The time model for matrix-matrix operations fails to predict matrix-vector operations accurately and results in large deviations for these benchmarks (up to 300 percent).

Table 1 gives the sizes of the data dependence graph and the lowered data dependence graph of the benchmark programs used in our experiments. As a further experiment, we wanted to investigate the quality of our heuristic scheduler by comparing the makespan of the heuristic with the makespan of the optimal solution. However, we were not able to compute an optimal solution for the problem sizes from Table 1 and had to reduce them by the order of three magnitudes. The largest problem was "reachability" with 53 nodes and 104 edges that was solved by CPLEX 10.0 [34] in approximately 41 minutes.

The makespan of the heuristic coincided with the makespan of the optimal solution. The time to compute the heuristic schedule was too small to be measured, which indicates that the heuristic is efficient and effective. However, the small problem instances of our benchmark may not offer enough degrees of freedom to see a gap between the heuristic and the optimal solution. Unfortunately, we were not able to increase the problem sizes without making the problems intractable to be solved by CPLEX.

In the remaining set of experiments, we investigated the extent to which our framework utilized the parallel execution units of the Cell. We have measured the execution time of the different phases of our framework across all benchmarks when run with eight SPEs. It follows that the majority of time (more than 75.59 percent) is spent executing matrix operations on Cell SPEs. The remaining 24.41 percent are mainly spent on scheduling (11.4 percent) and lowering of matrices into blocks (9.44 percent). Only 0.83 percent of time is spent on matrix allocation and 2.74 percent on deallocation and other cleanup.

Fig. 12 shows the time that SPEs spent idling during the execution phase of the system. The idle time is composed of the time the SPEs spent waiting for tasks to be offered by the PPE and the time spent waiting for the completion of DMA transfers. Again, these numbers were determined for eight SPEs. With the "dma" benchmark, we approached the Cell's peak performance, which is manifested in idle times close to zero.

TABLE 1
Number of Operations (Nodes) and Dependencies (Edges) in the Original and Lowered Dependence Graphs

	Original		Lowered	
	Nodes	Edges	Nodes	Edges
dft	9	9	72384	141284
dma	60003	120000	60003	120000
hill	9	8	73684	140608
hits	50	98	319950	639450
kmeans	34	23	137764	252743
leontief	204	402	129152	254720
markov	52	100	93400	186580
neural	10012	12489	231784	410151
reachability	960	1912	119600	239000

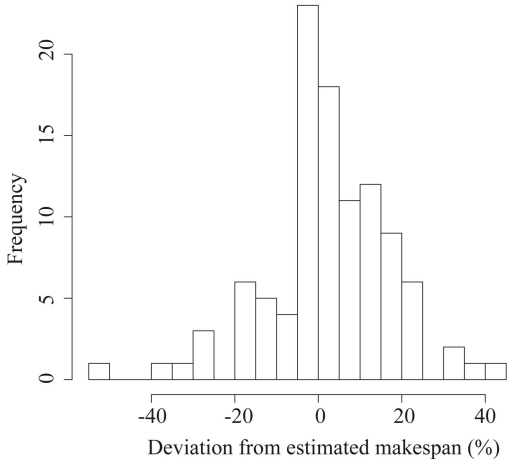


Fig. 11. Estimated versus observed makespans.

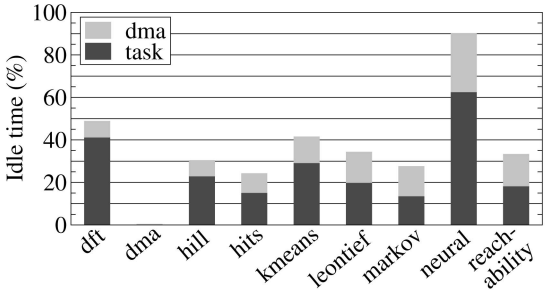


Fig. 12. SPE idle times.

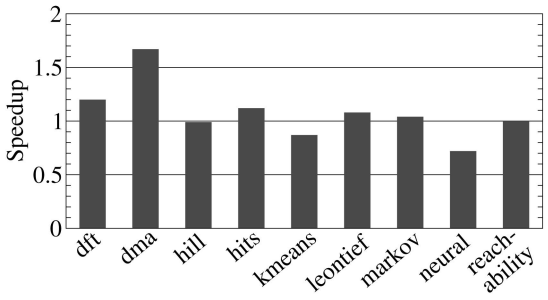


Fig. 13. Overlapped execution of interpreter with computation of matrices on SPE.

The remaining benchmarks incur idle times as a result of three main factors. First, operations such as matrix addition are memory-bound (rather than computationally bound). Although our triple buffering technique overlaps computation with communication, memory-bound operations can still induce DMA waiting times because memory transfers for these operations are more expensive than computation of the result.

The second reason for idle times is that certain traces may contain a large number of data dependencies that reduce the amount of parallelism, which can be exploited within the trace. This can result in idle time spent waiting for the operands of an operation to become available. Third, if operations are computed too quickly by the SPEs, the execution control mechanism may not be able to service them fast enough, resulting in time spent waiting for a task to be offered by the PPE. The impact of this could be reduced by aggregating several operations into a single operation, which takes a longer time to execute.

The graph in Fig. 13 shows the speedup in execution of each benchmark when execution of matrix operations is overlapped with execution of the Octave interpreter. When the trace of matrix operations, which is accumulated through lazy evaluation, reaches a threshold in size, execution is triggered even though a result in the trace is not needed by the Octave program. This allows computation of the matrix operations to be overlapped with continued execution of the Octave program by the interpreter. Most benchmarks achieve a speedup when compared with sequential execution of the Octave interpreter and our system. Those that do not, such as "kmeans" and "neural" may not produce traces that are large enough to allow any extra parallelism to be exploited and instead, the overhead of using several threads for execution causes a slowdown. Fine tuning of the threshold at which execution of a trace is triggered may result in further speedups from using this technique.

9 Conclusion

In this work, we have shown how to speed up the execution of matrix language scripts with modern accelerator architectures by taking advantage of

1. data parallelism,
2. instruction-level parallelism,
3. pipeline parallelism, and
4. task parallelism.

The execution of Octave scripts on our framework for the Cell Broadband Engine architecture is up to a factor of 12 faster than standard Octave on more recent Intel Core2 Quad processors. The parallelization of matrix language scripts is done transparently, i.e., no programmer intervention is required. Parallelization is mainly achieved by changing the strict sequential order of matrix language scripts to a partial order, allowing independent operations to be executed concurrently, as well as by exploiting the data parallelism of matrix operations.

Acknowledgments

The authors would like to thank Dr. Eduard Mehofer for his support and the access to the IBM Blade cluster of the Department of Scientific Computing at the University of Vienna.

References

[1] MathWorks, "MATLAB," http://www.mathworks.com.au/, 2010.
[2] GNU, "Octave," http://www.gnu.org/software/octave/, 2010.
[3] R. Choy and A. Edelman, "Parallel MATLAB: Doing It Right," *Proc. IEEE,* vol. 93, no. 2, pp. 331-341, Feb. 2005.
[4] J. Kepner, "MatlabMPI," *J. Parallel Distributed Computing,* vol. 64, no. 8, pp. 997-1005, 2004.
[5] J. Fernandez, M. Anguita, E. Ros, and J. Bernier, "SCE Toolboxes for the Development of High-Level Parallel Applications," *Lecture Notes in Computer Science,* p. 518, Springer, 2006.
[6] N.T. Bliss and J. Kepner, "pMATLAB Parallel MATLAB Library," *Int'l J. High Performance Computing Applications,* vol. 21, no. 3, pp. 336-359, 2007.
[7] N. Travinin, H. Hoffmann, R. Bond, H. Chan, J. Kepner, and E. Wong, "pMapper: Automatic Mapping of Parallel MATLAB Programs," *Proc. High Performance Computing Modernization Program Users Group Conf.,* pp. 254-261, 2005.
[8] G. Sharma and J. Martin, "MATLAB: A Language for Parallel Computing," *Int'l J. Parallel Programming,* vol. 37, no. 1, pp. 3-36, 2009.
[9] Interactive Supercomputing, "Star-P—Parallel Computing without Parallel Programming," White Paper, 2008.
[10] L. De Rose and D. Padua, "A MATLAB to Fortran 90 Translator and Its Effectiveness," *Proc. 10th Int'l Conf. Supercomputing,* pp. 309-316, 1996.
[11] R.C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," *Proc. SuperComputing: High Performance Networking and Computing,* 1998.
[12] AccelerEyes, "Accelereyes—MATLAB GPU Computing," http://www.accelereyes.com/, Mar. 2009.
[13] P. Messmer, P. Mullowney, and B. Granger, "GPULib: GPU Computing in High-Level Languages," *Computing Science Eng.,* vol. 10, no. 5, pp. 70-73, 2008.
[14] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, and J. Sancho, "Entering the Petaflop Era: The Architecture and Performance of Roadrunner," *Proc. ACM/IEEE Conf. Supercomputing,* 2008.
[15] M. Snir and S. Otto, *MPI—The Complete Reference: The MPI Core.* MIT Press, 1998.
[16] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Parallel Programming with Polaris," *Computer,* vol. 29, no. 12, pp. 78-82, Dec. 1996.
[17] G. Almasi and D.A. Padua, "MaJIC: A MATLAB Just-In-Time Compiler," *Languages and Compilers for Parallel Computing,* pp. 68-81, Springer, 2001.
[18] M.J. Quinn, A. Malishevsky, and N. Seelam, "Otter: Bridging the Gap between MATLAB and ScaLAPACK," *Proc. Seventh IEEE Int'l Symp. High Performance Distributed Computing (HPDC '98),* p. 114, 1998.
[19] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, "A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* pp. 39-48, 2000.

[20] AccelerEyes, "Jacket User Guide," http://www.accelereyes.com/, Mar. 2009.
[21] The GP-you Group, "GPUmat: GPU Toolbox for MATLAB," http://gp-you.org/, Oct. 2009.
[22] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys,* vol. 21, no. 3, pp. 359-411, 1989.
[23] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach,* fourth ed. Morgan Kaufmann, 2007.
[24] R. Khoury, B. Burgstaller, and B. Scholz, "Accelerating the Execution of Matrix Languages on the Cell Broadband Engine Architecture," arXiv.org, vol. arXiv:0910.2324v2 [cs.PL], http://arxiv.org/abs/0910.2324v2, Nov. 2009.
[25] M.R. Garey and D.S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W.H. Freeman & Co., 1990.
[26] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys,* vol. 31, no. 4, pp. 406-471, 1999.
[27] R.H. Möhring, M.W. Schäffter, and A.S. Schulz, "Scheduling Jobs with Communication Delays: Using Infeasible Solutions for Approximation," *Proc. Fourth European Symp. Algorithms,* pp. 76-90, 1996.
[28] E. Kreyszig, *Advanced Engineering Mathematics,* ninth ed. Wiley, Nov. 2005.
[29] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro,* vol. 26, no. 2, pp. 10-24, Mar. 2006.
[30] R.E. Bryant and D.R. O'Hallaron, *Computer Systems: A Programmer's Perspective.* Prentice Hall, 2003.
[31] W.P. Adams and R.J. Forrester, "A Simple Recipe for Concise Mixed 0-1 Linearizations," *Operations Research Letters,* vol. 33, no. 1, pp. 55-61, 2005.
[32] IBM, *Cell Broadband Engine Programming Handbook.* IBM, May 2008.
[33] IBM, *Cell SDK Example Library API Reference 3.1.* IBM, Sept. 2008.
[34] Ilog Inc., "Solver Cplex," http://www.ilog.fr/products/cplex/, 2003.

Raymes Khoury received the BCST (Adv) and BCST (Hons) (Adv) degrees from The University of Sydney, in 2009. He is an author and coauthor of publications in parallel computing and wireless sensor networks. He is currently with the School of Information Technologies, University of Sydney. His research interests include parallel computing on multicore architectures, query systems for wireless sensor networks, and data visualization.

Bernd Burgstaller received the PhD degree in static program analysis at the Vienna University of Technology in 2005 and continued as a postdoctorate at The University of Sydney until 2007. He is currently an assistant professor in the Department of Computer Science at Yonsei University. Before pursuing an academic career, he worked in industry as a programmer and software architect for Philips Consumer Electronics, Vienna.

Bernhard Scholz received the Dipl.-Ing (eq. MEng) and PhD degrees from Vienna University of Technology, Austria, in 1997 and 2001, respectively. He is currently a senior lecturer in the School of Information Technologies at The University of Sydney. His research interests include programming languages and parallel and embedded systems.