Institutionen för datavetenskap
Department of Computer and Information Science

Master's Thesis

A Skeleton Library for Cell Broadband Engine

Markus Ålind

Reg Nr: LITH-IDA-EX--08/002--SE
Linköping 2008

Linköpings universitet
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköpings universitet
SE-581 93 Linköping, Sweden

Institutionen för datavetenskap
Department of Computer and Information Science

Master's Thesis

A Skeleton Library for Cell Broadband Engine

Markus Ålind

Reg Nr: LITH-IDA-EX--08/002--SE
Linköping 2008

Supervisor: **Mattias Eriksson**
IDA, Linköpings universitet

Examiner: **Christoph Kessler**
IDA, Linköpings universitet

Department of Computer and Information Science
Linköpings universitet
SE-581 93 Linköping, Sweden

Avdelning, Institution
Division, Department

The Programming Environments Laboratory, IDA
Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Datum
Date

2008-03-07

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
__

ISRN
LITH-IDA-EX--08/002--SE

Serietitel och serienummer **ISSN**
Title of series, numbering __

URL för elektronisk version

Titel
Title

Ett Skelettbibliotek för Cell Broadband Engine
A Skeleton Library for Cell Broadband Engine

Författare
Author

Markus Ålind

Sammanfattning
Abstract

The Cell Broadband Engine processor is a powerful processor capable of over 220 GFLOPS. It is highly specialized and can be controlled in detail by the programmer. The Cell is significantly more complicated to program than a standard homogeneous multi core processor such as the Intel Core2 Duo and Quad. This thesis explores the possibility to abstract some of the complexities of Cell programming while maintaining high performance. The abstraction is achieved through a library of parallel skeletons implemented in the bulk synchronous parallel programming environment NestStep. The library includes constructs for user defined SIMD optimized data parallel skeletons such as map, reduce and more. The evaluation of the library includes porting of a vector based scientific computation program from sequential C code to the Cell using the library and the NestStep environment. The ported program shows good performance when compared to the sequential original code run on a high-end x86 processor. The evaluation also shows that a dot product implemented with the skeleton library is faster than the dot product in the IBM BLAS library for the Cell processor with more than two slave processors.

Nyckelord
Keywords

NestStep, Cell, BlockLib, skeleton programming, parallel programming

Abstract

The Cell Broadband Engine processor is a powerful processor capable of over 220 GFLOPS. It is highly specialized and can be controlled in detail by the programmer. The Cell is significantly more complicated to program than a standard homogeneous multi core processor such as the Intel Core2 Duo and Quad. This thesis explores the possibility to abstract some of the complexities of Cell programming while maintaining high performance. The abstraction is achieved through a library of parallel skeletons implemented in the bulk synchronous parallel programming environment NestStep. The library includes constructs for user defined SIMD optimized data parallel skeletons such as map, reduce and more. The evaluation of the library includes porting of a vector based scientific computation program from sequential C code to the Cell using the library and the NestStep environment. The ported program shows good performance when compared to the sequential original code run on a high-end x86 processor. The evaluation also shows that a dot product implemented with the skeleton library is faster than the dot product in the IBM BLAS library for the Cell processor with more than two slave processors.

Acknowledgments

I would like to thank my examiner Christoph Kessler and my supervisor Mattias Eriksson for their time, ideas and advices during this project. I would also like to thank Daniel Johansson for answering my questions on his NestStep implementation and Cell programming in general, Matthias Korch and Thomas Rauber for letting me use their ODE solver library and Pär Andersson at National Supercomputing Centre in Linköping for running some benchmarks on a few machines in their impressive selection.

Contents

Chapter 1

Introduction

1.1 Introduction

The fact that the superscalar processor's performance increase has slowed down and probably will slow down even more over the next few years has given birth to many new multi core processors such as the Intel Core2 dual and quad core and the AMD x2 series of dual core processors. These are all built of two or more full blown superscalar processor cores put together on a single chip. This is convenient for the programmer as they act just like the older symmetrical multi processor systems that have been around for many years. IBM, Sony and Toshiba have recently contributed to the multi core market with their *Cell Broadband Engine* which is a heterogeneous multi core processor. The Cell consists of one multi threaded Power PC (the *PPE*, PowerPc Element) core and eight small vector RISC cores (the *SPE*:s, Synergistic Processing Element). The Cell has a peak performance of 204 GFLOPS for the SPE:s only and an on chip communication bus capable of 96 bytes per clock cycle. The architecture allows the programmer very detailed control over the hardware. For the same reason the architecture also demands a lot from the programmer. To achieve good performance the programmer has to take a lot of details in consideration. A modern X86 processor will often get decent performance of mediocre or even bad code. The Cell will not perform well unless the programmer puts due attention to the details.

A large part of what makes it difficult and time consuming to program the Cell is memory management. The SPE:s does not have a cache like most processors. Instead of a cache all SPE:s have their own explicitly managed piece of on chip memory called *local store*. As the SPE local store is very small very few data sets will fit into it. Because of this almost all data sets has to be stored in the main memory and then moved between main memory and local store constantly. Memory transfers are done through DMA. On top of that, memory transfers have to be double buffered to keep the SPE busy with useful work while the memory flow controller (*MFC*) is shifting data around.

The *NestStep* programing environment helps with some of the memory management but the main hassle is still left to the programmer to handle. In this

master thesis we examine a skeleton approach on making it easier to program for the Cell processor.

1.2 Project Goals

This master thesis project aims to make it easier to write efficient programs for the Cell BE within the NestStep environment. The approach is to try to do this through a library of parallel building blocks realized as parallel skeletons. The aim is to cover memory management and parallelization as much as possible and help with *SIMD* optimization to some extent. This should also make the developed program less Cell specific and easier to port to other NestStep implementations on different types of hardware. The aim is to achieve this while maintaining good performance.

1.3 Project Approach

To learn more about the Cell processor the first step of the project was to design and implement a library covering a small part of BLAS. The library would be called from ordinary PPE code and the library would handle everything concerning parallelization and memory management. During the implementation of this library IBM released a complete PPE-based BLAS library as part of their Cell SDK 3.0. This made it unnecessary to finish the BLAS part of the project as IBM's library was far more complete than our version was supposed to become.

The next step of the project was to familiarize ourselves with the NestStep environment and the NestStep Cell runtime system. By doing so we could understand the obstacles for NestStep programming on Cell. The skeletons were then created to take care of these obstacles while optimizing performance at the same time.

The skeleton library that was created was then evaluated by small synthetic benchmarks and, by porting a computational PC application to NestStep and the Cell using the library. One of the synthetic benchmarks implements a dot product and was compared to the performance of the dot product in the IBM BLAS library for the Cell processor.

1.4 Implementation Approach

The implementation had two main goals. The first is usability: it is difficult and cumbersome to write fast code for the Cell. The second is performance: good usability is useless unless the result is fast enough. If performance was not an issue one would write an ordinary sequential PC program instead.

The basic skeletons took longer to finish than anticipated because of the amount of details that needed attention to achieve good performance. As the library was optimized it became clear that the NestStep synchronization system was too slow. The skeleton execution time was too short and put too much pressure

on the NestStep runtime system. To solve this a new streamlined inter SPE synchronization a combining system was developed.

1.5 Thesis Outline

The thesis is outlined as follows:

- Cell BE Processor Overview: An Overview of the Cell BE processor. Covers the architecture and performance of the processor. This chapter also covers how to write fast code in detail.
- NestStep Overview: An Introduction to the NestStep Language and runtime systems.
- Skeleton Programming: A short introduction to skeleton programming.
- BlockLib: This chapter covers the functionality and the implementation of the library that is the result of the findings of this project.
- Evaluation: This chapter covers the evaluation of BlockLib. The evaluation consists of both synthetic performance benchmarks and a usefulness test in which a real application is ported to NestStep and the Cell using BlockLib.
- Conclusion and Future Work: Here are the overall results of the project discussed. The chapter also covers what could be done in future work on the subjects, such as expansions and improvements.
- Glossary: Domain specific words and abbreviations are explained here.
- BlockLib API Reference: A short API listing and reference from the BlockLib source distribution.
- Code for Test Programs: The code used in the evaluation chapter are found here. This chapter includes the core routines from the real application port.

The most important contributions of this thesis are also described in a conference paper [1].

Chapter 2

Cell BE Processor Overview

The Cell Broadband Engine is a processor developed by IBM, Sony and Toshiba. Most modern high performance multi core processors are homogeneous. These homogeneous multi core processors are built of two or more identical full blown superscalar processors and act like a normal multi processor SMP machine. The Cell BE on the other hand is a heterogeneous processor built of a normal PowerPC core (PPE) and eight small vector RISC processors (SPE:s) coupled together with a very powerful bus. The processor is designed to enable inter chip cooperation as opposite to most other multi core processors which are designed to act more as independent processors. The Cell does not share the abstraction layers of the normal x86 or PowerPC processors, it is striped of most advanced prediction and control logic to make room for more computational power. The result is extreme floating point performance and programmer control. See Figure 2.1 for the overall architecture.

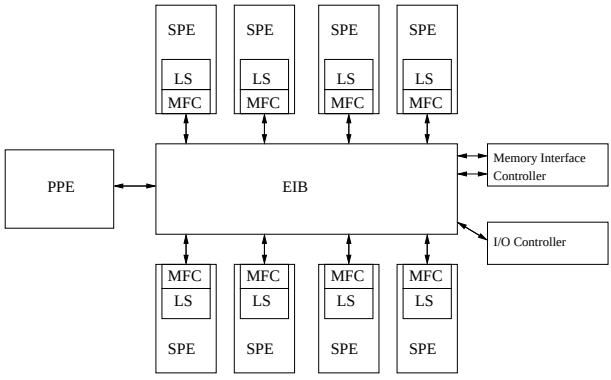

Figure 2.1. Cell BE architecture

2.1 Literature on Cell BE

"Cell Broadband Engine Architecture" [5] is an extensive document on the Cell BE architecture. "Cell BE Programming Tutorial" [8] largely covers the same subjects but is more focused on programming. Both documents are big, more than 300 pages each. "Introduction to the Cell multiprocessor" is a compact overview focused on the actual architecture in just 16 pages and is suitable as a primer.

2.2 EIB

The Element Interconnect Bus (*EIB*) connects the PPE, the SPE:s, the memory controller and IO controller. The bus is built of four rings and is capable of 8 bytes per bus unit per clock cycle. The twelve bus units (the memory controller has two bus connections) brings the EIB total capacity up to 96 bytes per clock cycle. Each SPE has an EIB bandwidth of 25.6 GB/s but the actual throughput is dependent on which SPE that communicate with which [5].

2.3 SPE

The SPE:s are vector processors. That means that they can perform calculations on vectors of operands. The vectors on the SPE:s are 16 bytes which translate to four single precision floats or two double precision floats. The SPE can issue a single precision vector operation each clock cycle and this is the key to the high performance. The whole infrastructure of the SPE is built for 16 byte vectors. The SPE always loads and stores 16 bytes at a time, even if it only needs a scalar.

Each SPE has 256 kiB of fast memory called *local store*. The local store is the SPE's working RAM and all data and code has to fit into it. The system main memory is not directly accessible. Data is transferred between system main memory and the local stores using DMA.

2.4 Programs

The SPE:s and the PPE have their own instruction sets. SPE programs are compiled with a different compiler than the PPE programs. All normal programs, such as the operating system and most user programs, such as the shell, run on the PPE. The PPE runs normal PowerPC code in 64 or 32 bit mode. SPE programs are managed by the PPE program as a special type of threads. A usual approach is to use the PPE program for initialization and administration and the SPE:s to do the heavy calculations with the PPE program as a coordinator.

2.5 Performance

The Cell BE has a peak single precision performance of over 25.6 GFLOPS for each SPE. This sums up to 204 GFLOPS excluding the PPE. The double precision

performance is 1.83 GFLOPS per SPE or 14.63 GFLOPS in total [18].

2.6 PlayStation 3

The Sony PlayStation 3 is by far the cheapest Cell hardware. It can be bought for $400 (as of December 2007, Wal Mart Internet store). The PlayStation is a gaming console but Sony has prepared it to run other operating system in a hypervisor. It is relatively easy to install Linux and it runs several distributions such as Fedora Core, Ubuntu, Gentoo and Yellow Dog Linux.

2.6.1 Limitations

The supervisor restricts the access to some of the hardware, such as the graphic card and hard drive. The PS3 is very limited on some major points. It has only just above 210 MiB of RAM available inside the hypervisor. The hypervisor also occupies one SPE. One other SPE is disabled altogether. All this limits the applications running in Linux inside the hypervisor to under 200 MiB of RAM and six SPE:s. The total performance for all SPE:s is 153 GFLOPS in single precision and 11 GFLOPS in double precision.

2.7 IBM QS21

IBM QS series is IBM professional Cell hardware for computation clusters. The blades fit in the BladeCenter H chassis. Each 9 unit cabinet holds up to 14 blades. The current blade is the QS21. This blade has two 3.2 GHz Cell BE and 2 GiB of ram. This sums up to over 6 TFLOPS single precision performance and 28 GiB of ram in 9 units of rack space [10].

2.8 Writing Fast Code for Cell BE

While it is possible to achieve close to peak performance for some real applications, there are a lot of major and minor issues to deal with. An ordinary generic C-implementation of a given algorithm will perform poorly. Most tips in this chapter are also covered by Brokenshire in a paper on cell performance [2]. The tips are based on or verified by experiments and benchmarking done in this project.

2.8.1 Floats, Doubles and Integers

The SPE can issue one float vector operation per cycle or one double vector operation every seventh cycle. As the float vector is four elements and the double vector only two elements the SPE:s are fourteen times faster when calculating single precision operations than on double precision operations. This means that doubles should be avoided wherever the extra precision is not absolutely necessary.

The SPE does not have a full 32 bit integer multiplier but just a 16 bit multiplier. This means that 16 bit integers (shorts) should be used if possible.

2.8.2 Memory Transfers

Alignment with DMA

For DMA transfers, in most cases data have to be 16 byte aligned. If a DMA-transfer is bigger than eight bytes it has to be a multiple of 16 bytes in size and both source and target address have to be 16 byte aligned. On less than 8 byte transfers both source and target have to have the same alignment relative to a 16 byte aligned address. To achieve maximal memory bandwidth both source and target address have to be 128 byte aligned. If target or source address is not 128 byte aligned the transfer will be limited to about half the achievable memory bandwidth. This is crucial as memory bandwidth often is a bottleneck. With GCC the `__attribute__((aligned(128)))` compiler directive does not work with memory allocated on the stack. The alignment is silently limited to 16 bytes. By allocating a 128 bytes bigger array than needed a 128 byte aligned start point can be found.

Multi Buffering

All transfers between main memory and local store are done through explicit asynchronous DMA transfers. This means that it is possible to issue DMA transfers to one buffer while performing calculations on another. This double buffering can hide memory transfers completely if computation of a block takes longer than transferring it to or from main memory. If the data chunk is to be written back to main memory, double buffering should be used for those transfers as well.

Multiple SPE:s in Transfer

A single SPE can not fully utilize the available main memory bandwidth. A set of synthetic memory benchmarks by Sándor Héman et al. [4] show that a single SPE can achieve about 6 GB/s in a benchmark where six SPE will achieve about 20 GB/s. Even as each SPE's connection to the EIB is capable of 25 GB/s this speed is not possible to and from main memory which has higher latency relative to communication that stays on chip.

Main memory usage should be distributed over three or more SPE:s if possible to avoid a performance penalty from this limitation.

2.8.3 SIMD

The SPE is a SIMD processor. The whole processor is built around 128 bit vectors which hold four floats, four 32 bit integers or two doubles.

C Intrinsics

To make it easier to use the SIMD instructions on the SPE:s and PPE there exist a large set of language extensions for C and C++. These intrinsics map to SIMD instructions but let the compiler handle the registers, loads and stores. This makes it much easier to write efficient SIMD code and should be much less error prone.

It also lets the compiler do optimizations such as instruction scheduling as usual. All intrinsics are specified and explained in the Cell language extension reference [7].

SIMD Math Library

The SIMD instructions cover just the very basic arithmetical operations. IBM ships a math library in recent CELL SDK which is similar to the standard C math.h library but operates on vectors. This library gives access to a set of standard arithmetic operations that have been SIMD optimized and fits well with the SIMD Intrinsics. See the SIMD Math library reference [9] for a complete specification.

Alignment with SIMD

With SIMD instructions the data (organised as vectors) should be 16 byte aligned to avoid any shifting and extra loads. All loads from and stores to local store is 16 byte aligned quad words. A whole quad word is loaded even if the requested data is a single byte or a 32 bit integer. If a quad word that is not 16 byte aligned is requested two quad words have to be loaded. The wanted quad word is then selected, shifted and combined to fit into a vector register. Scalar arithmetics is in fact often slightly slower than arithmetics on a vector of four floats. This is partly because the scalar often is not 16 byte aligned so it has to be moved to the preferred scalar slot. This also implies that even scalars should be 16 byte aligned if possible.

2.8.4 Data Dependencies

While the SPE can issue one vector float operation every cycle the result of the operation is not in its target register until six cycles later. If the instruction after depends on that result the SPE has to be stalled until it is ready. If the C intrinsics are used the compiler handles the instruction scheduling to minimize stalled cycles. The programmer still has to implement his algorithm in such way that there is enough independent computations to fill the pipeline, if possible.

2.8.5 Striping of Binaries

The small local store makes it easy to run out of SPE memory. Code, variables, buffers etc. has to fit into the 256 kiB. There is no mechanism to protect the stack from colliding with something else. The program continues to run, sometimes almost correctly, often producing very odd bugs. A way to avoid this is to keep an eye on the SPE binary and make sure that there is a margin between binary size plus estimated SPE memory usage and 256 kiB. Make sure that the SPE binary is striped before it is embedded into a PPE object file. Striping will prevent debugging in GDB but it will also shrink the binary considerably.

Chapter 3

NestStep Overview

The bulk synchronous parallel (*BSP*) programming model divides the program into a series of *supersteps*. Each superstep consist of a local computation part, a group communication part and a synchronization barrier. The processors work individually in the local computation part and then perform necessary communication. The communication can be to share data with other processors or to combine a result to a total result. See Figure 3.1 for program flow.

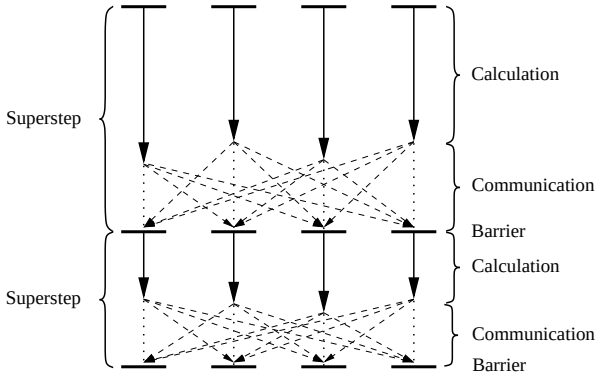

Figure 3.1. Bulk synchronous parallel execution flow.

NestStep is a language that implements the BSP model. It has a shared memory abstraction for distributed memory systems. It has support for shared variables and arrays and uses the BSP superstep in its memory consistency model, all shared data are synchronized in the BSP combine part of each superstep. This means that shared data consistency is only guaranteed between each superstep.

3.1 Literature on NestStep

The NestStep language and structure is covered in "NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model" [15]. More details on memory management on distributed memory systems with NestStep is covered in "Managing distributed shared arrays in a bulk-synchronous parallel programming environment: Research Articles" [14].

The NestStep implementation for the Cell BE used in this master project is written by Daniel Johansson in his master's project where he ported NestStep runtime system to the Cell processor. Johansson discusses the implementation details and gives a NestStep overview in his thesis "Porting the NestStep Runtime System to the CELL Broadband Engine" [11]. Daniel Johansson's Cell port is based on an implementation for PC clusters written by Joar Sohl in his master project [17].

3.2 Variables and Arrays

NestStep has support for shared and private variables and arrays. The private variables and arrays behave just like ordinary C data types and are accessible only by the processor that owns them. The shared variables are synchronized after each superstep.

NestStep also has support for two distributed data types: block distributed arrays and cyclic distributed arrays. The block distributed array is distributed across the processors, each processor is assigned a contiguous part of the array. The distributed cyclic array is partitioned in smaller chunks which are assigned to the processors in a cyclic manner. The different array partitions provide a implicit partition of the data set for simple parallelization. If the computations are heavier in some part of an array the cyclic partition can help to load balance the chunks better than the block distributed array. See Figure 3.2 for a figure of array partitioning with NestStep distributed arrays.

3.3 Combine

The shared variables and arrays are combined after each superstep, i.e. they are synchronized. The combining makes sure that all copies of a variable are consistent over all processors. The shared data is combined in a deterministic programmable way. Variables can be combined by a programmable reduction, like global `MAX` or a global `SUM`. The combine can also set all copies of a variable to the value of a certain node's copy. A shared array is combined element by element and thus behaves like an array of individual shared variables when combined.

3.4 NestStep Implementations

The first NestStep implementations were developed for PC clusters on top of MPI such as Sohls [17]. MPI stands for message passing interface and is an interface for

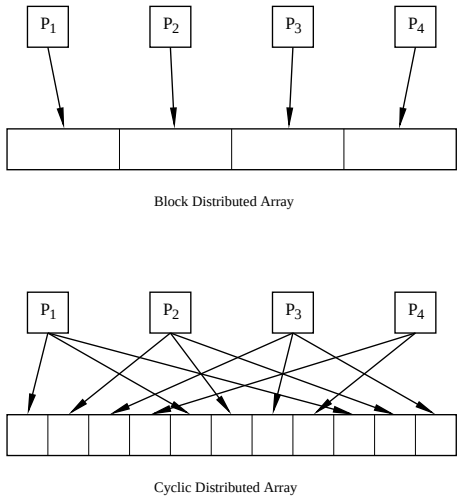

Figure 3.2. Array partitioning with NestStep distributed arrays

sending messages and combining and spreading data between processors on shared memory systems (such as multi processor PCs) and distributed memory systems (such as PC clusters). NestStep was then ported to the Cell by Daniel Johansson [11].

3.4.1 Cell BE NestStep Implementation

The NestStep Cell port is extended compared to the MPI based versions to handle some of the difference between a PC cluster and a Cell BE. All user code runs on the SPE:s but the runtime system runs on both the PPE and the SPE:s. The PPE act as the coordinator, manages SPE threads and combines. The SPE part of the runtime system handles communication with the PPE and main memory transfers.

Variables and Arrays

All shared data are resident in main memory. This is mainly because there is too little space in the local store. To handle this the Cell NestStep implementation is extended with some extra variable and array handling and transferring functionality. Small variables are transferred between main memory with explicit get and store functions. As whole arrays often do not fit in the local store, they can be transferred in smaller chunks. There are also private variables and arrays in main memory, which are managed in the same way as the shared variables and arrays, except that they are not touched by combine. For this project we are especially interested in the NestStep *block distributed* arrays.

Combine

The Cell NestStep combine is handled by the PPE. All changes to shared data made by SPE:s have to be transferred to main memory prior combining. The combine with global reduction is limited to a few predefined functions in the Cell port [11].

Chapter 4

Skeleton Programming

Skeletons are generic structures that implements a computation pattern rather than a computation itself. The skeleton can also hide the implementation behind an abstract interface. Skeletons are often derived from higher order functions such as *map* from functional languages. Map is a calculation pattern which applies a function on each of the elements in an array. The user provides the map with a function and an array and does not have to know nor care about how the calculation is performed. The library or compiler that provides the map skeleton can implement it in the most efficient way on the current platform. It may even run a different implementation depending on some runtime parameter such as the number of elements in the array.

4.1 Literature on Skeletons

"Practical PRAM Programming" [13] is a book covering many aspects of the PRAM (parallel random access machines) computers, the FORK programming language and related subjects. Chapter 7 is an overview of parallel programming with skeletons which is particularly interesting for this project. Murray I. Cole has written several books and papers on the subject of parallel skeleton programing, such as "Algorithmic Skeletons: Structured Management of Parallel Computation" [3].

4.2 Code Reuse

Skeletons are also an implementation of code reuse which can cut developing time considerably. A skeleton can be used instead of a whole set of library functions. For instance, instead of implementing summation of an array, max of an array, min of an array etc. a *reduce* skeleton can be provided. The user then provides the skeleton with the reduction function. A sum is a reduce with `ADD` as the reduction function. This way a whole group of functions can be replaced by a single skeleton. This will also let the user reduce arrays with more uncommon functions, which

probably would not be available without skeletons, such as finding the elements with the smallest absolute value.

4.3 Parallelization

A skeleton can be realized as a parallel computation even when the interface looks like it is sequential [13]. The interface can thereby be identical on several different platforms, single processor, multi processor and even distributed memory systems such as PC clusters. The implementation can the be optimized for the specific platform, the number of available processors etc. This will also limit the platform specific code to the skeleton.

Chapter 5

BlockLib

BlockLib is the result of this master project. The library tries to ease Cell programming by taking care of memory management, SIMD optimization and parallelization. BlockLib implements a few basic computation patterns as skeletons. Two of these are map and reduce which are well known. BlockLib also implements two variants of those. The first is a combined map and reduce as a performance optimization and the other is a map that enables calculation of one element to access nearby elements. The parallel skeletons are implemented as a C library. The library consists of C code and macros and requires no extra tools besides the C preprocessor and compiler.

The BlockLib skeleton functions are their own NestStep superstep. The map skeleton can also be run as a part of a bigger superstep.

5.1 Abstraction and Portability

The usage of BlockLib does not tie the user code to the Cell platform. The same interface could be used by an other BlockLib implementation on an other NestStep platform in an efficient way, with or without SIMD optimization. BlockLib can serve as an abstraction of Cell programming to a less platform specific level.

5.2 Block Lib Functionality

5.2.1 Map

The skeleton function *map* applies a function on every element of one or several arrays and stores the results in a result array [12]. The result array can be one of the argument arrays. The skeleton can also be described as $\forall i \in [0, N-1], r[i] = f(a_o[i], \ldots, a_k[i])$. The current implementation is limited to three argument arrays.

5.2.2 Reduce

Reduce reduces an array to a scalar by applying a function on one element of the argument array and the reduced result of the rest of the array recursively. A reduce with the function *ADD* sums the array and a reduce with *MAX* finds the biggest element. A reduction with the operation op can be described as $r = a[0]$ op $a[1]$ op ... op $a[N-1]$. This implementation requires the applied function to be associative [12]. Some reductions will have a result that varies with the computation order when applied to large arrays with reduce, even if they are associative. This is caused by the limited floating point precision of floats and doubles and is not an error of the skeleton function. For instance, the result of a summation over a large array will vary with the number of SPE:s used.

5.2.3 Map-Reduce

Map-reduce is a combination of map and reduce. The map function is applied on every element before it's reduced. The result is the same as if map is first applied to an array a, producing the result b on which reduce is then applied. This can also be described as $f(a_o[1], \ldots, a_k[1])$ op $f(a_o[2], \ldots, a_k[2])$ op ... op $f(a_o[N-1], \ldots, a_k[N-1])$. This way the result after map do not have to be transferred to main memory and then transferred back again for reduction. This reduction of main memory transfers improves performance as main memory bandwidth is a bottleneck. The combination will also save main memory by making the array b redundant.

5.2.4 Overlapped Map

Some calculation patterns are structurally similar to a map but the calculation of one element uses more elements than the corresponding element from the argument array. This can be described as $\forall i \in [0, N-1], r[i] = f(a[i-k], a[i-k+1] \ldots, a[i+k])$. One such common calculation is convolution. Many of those calculations have a limited *access distance*. A limited access distance means that all needed argument elements are within certain number of elements from the calculated element. A one dimensional discrete filter has an access distance limited by the filter kernel size (the number of non zero coefficients) for example.

The *overlapped map* can either be cyclic or zero padded. A read outside the array bounds in a cyclic overlapped map will return values from the other end of the array, i.e. for an array a of size N $\forall i \in [-N, -1], a[i] = a[i+N]$ and $\forall i \in [N, 2N-1], a[i] = a[i-N]$. The same read in a zero padded overlapped map will return zero, i.e. $\forall i \notin [0, N-1], a[i] = 0$.

The overlapped map's macro generated SIMD optimized code takes some performance penalty from not being able to load the operands as whole 16 byte vectors. The vector has to be loaded one operand at a time as they can have any alignment.

5.2.5 Miscellaneous Helpers

BlockLib contains some other functionalities over the skeletons such as memory management helpers, message passing primitives, and timers. Some of those are tied to the Cell architecture, other does not conform to the BSP model and should be used with care. They are discussed in the implementation section together with the technical problem they are designed to solve.

Pipeline

In some types of parallel computations, such as matrix-matrix or vector-matrix multiplication, some of the data is needed by all processors. This shared data can either be read from main memory by all processors individually or replicated between the processors. The Cell processor has a lot more bandwidth internally between SPE:s than main memory bandwidth. This can be used to enhance the total amount of operands being transferred through the SPE:s above the main memory bandwidth.

A *pipe* architecture can have an arbitrary layout but in its most basic layout each data chunk travels through the pipeline one SPE at a time. This way all SPE:s can get the shared data without occupying the main memory bus more than necessary. If the shared data was broadcasted to all SPE:s for each chunk the SPE that loaded the chunk from main memory would be a bottleneck. With larger data sets the pipe initialization time becomes negligible.

If the data flow layout in Figure 5.1 is used each SPE loads non shared data from main memory in parallel combined with the pipe and effective bandwidth can be maximised. If the shared and the non shared data chunks have the same size then the actual used memory bandwidth B_a is $B_a = N(p+1)/t_e$ (where p is the number of SPE:s and t_e the execution time) as $p+1$ arrays of equal size are read from memory. The effective bandwidth usable by the SPU:s B_e is $B_e = N(p+p)/t_e$. With six SPU:s $B_a/B_e = 1.714$ which would lead to significant performance improvements in applicable cases were memory bandwidth is a bottleneck. This setup is realized by the code in Listing 5.1.

The maximum memory bandwidth achieved with the BlockLib helper functions reading from a distributed array is 21.5 GB/s with 6 SPE:s. The code in Listing 5.1 achieved 20.1 GB/s in actual used memory bandwidth. The total amount of data transferred through all SPE:s is 34.4 GB/s in total which is considerably higher.

The pipe approach has several drawbacks. As described in 2.8.2 each SPE cannot maximise the main memory bandwidth alone. A single simple linear pipeline will therefore perform badly bandwidth wise. The pipeline usage presented above works around this problem but this pattern only fits a small number of computation patterns and data set sizes and formats. The alignment requirements on DMA transfers limit the usable data set formats even more. The pipeline introduces a synchronisation for each data chunk. Each SPE has to synchronize with its pipeline neighbours to make sure that they are ready for the next DMA transfer cycle.

These drawbacks would make a full blown pipe skeleton like the map and the reduce useless for all but a very small set of problems. Instead of a such skeleton the pipe helper functions can be used. They are more generic than a pure skeleton and should fit a larger set of problems and computation patterns. The pipe helper functions can be used to set up pipelines of arbitrary layout as long as there is not more than one pipeline link between two SPU:s in each direction. These functions will tie the code hard to the Cell architecture. The chunk size has to be less or equal to 4096 elements with single precision and 2048 elements with double precision. It also has to be a multiple of 128 bytes, i.e. 32 elements with single precision and 16 elements in double precision to achieve maximal performance. The functions will however work with chunk sizes which are a factor of 16 bytes (four single precision, two double precision) but at a price of a big performance degradation. A more advanced implementation could be made to handle arbitrary chunk sizes but this would lead to even worse performance.

Listing 5.1. Pipe test code.

```
// private array parr is of size N
// block distributed array x is of size p*N where p is the number of
// processors (SPE:s) in the group.
NestStep_step();
{
    pipeF_Handler pipe; // pipe handler
    BArr_Handler baX; // block dist array handler

    // set the pipe up.
    if(rank == 0)
        init_pipeF_first(&pipe, parr, rank+1, 4096);
    else if(rank != groupSize -1)
        init_pipeF_mid(&pipe, rank-1, rank+1, 4096);
    else
        init_pipeF_last(&pipe, NULL, rank-1, 4096);

    // init block dist array handler
    init_BArr_Get(&baX, x, 4096);

    while(step_PipeF(&pipe)) // loop trough all chunks
    {
        get_sw_BArr(&baX);
        // work on chunks here
        do_some_work(pipe.current.p, baX.current, pipe.current.size);
    }
    NestStep_combine(NULL,NULL);
}
NestStep_end_step();
```

5.3 User Provided Function

All the skeletons work by applying a *user provided function* on the elements of one or more arrays. There are several possible approaches to this and the results differs very much in performance and ease of use.

5.3.1 Simple Approach

The naive approach in C is to use function pointers for the user provided function and apply the function on each element in the array one element at a time. This is convenient for the user as the user provided function becomes very simple. The drawback is that this approach has an devastating effect on performance. The

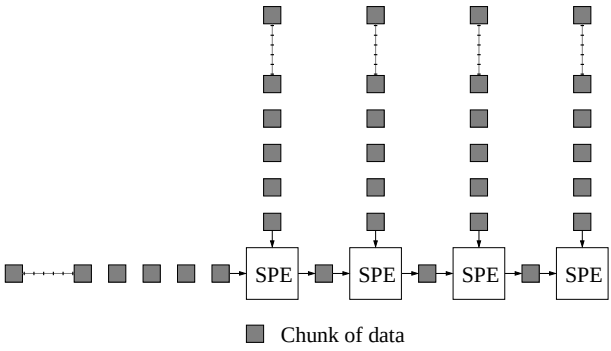

Figure 5.1. Data flow in a pipe construct.

function call via a function pointer is very slow and prevents performance boost from loop unrolling and auto vectorization. This method is useful if the function is very computation heavy or if the number of elements is small which makes the performance penalty impact on over all execution time negligible. See Listing 5.2 for an usage example using this method.

Listing 5.2. Example of skeleton usage, simple approach.

```
// Definition
float add(float left, float right)
{
    return left+right;
}
// Usage
res = sDistReduce(&add, x, N);
```

5.3.2 User Provided Inner Loop

One of the main purposes of the general map and reduce constructs is to spare the programmer from the burden of memory management on the Cell. The inner loop of the map and reduce constructs operate on smaller chunks of the arrays that has been transferred into the local store. These inner loops are not necessarily more complicated on the cell than on any other processor. If the user provides the construct with a complete *inner loop* the performance increases by several orders of magnitude for simple operations like addition. The number of function calls via function pointer is reduced from one per element to one per chunk and loop unrolling is possible. Chunks in BlockLib are 16 kiB (4096 floats or 2048 doubles) which works well with the Cell DMA system and local store size. Smaller chunks reduce bandwidth utilization and bigger increase SPE local store consumption without any bandwidth improvement. See Listing 5.3 for a usage example using this method.

Listing 5.3. Example of skeleton usage, inner loop approach.

```
// Definition
float addReduce(float *x, int n)
{
    int i;
    float res=0;
    for(i=0;i<n;i++)
    {
        res+=x[i];
    }
    return res;
}
// Usage
res = sDistReduceLocalFunc(&addReduce, x, N);
```

5.3.3 SIMD Optimization

To get even remotely close to the Cell peak performance the use of SIMD instructions is absolutely necessary. It is often faster to calculate four floats with SIMD instructions than to calculate a single float with non SIMD instructions. This means that a SIMD version of a function may achieve a speedup of over four. See Section 2.8.3 for more information on the SIMD issue.

The approach with a user provided inner loop enables the library user to SIMD optimize the program. The process of hand SIMD optimization of a function is a bit cumbersome and ties the code hard to the Cell. It also requires the programmer to have much knowledge of the Cell architecture. The example in Listing 5.4 shows a usage example that uses a hand optimized SIMD inner loop. This example and the one in Listing 5.3 compute the same thing. The only difference is the SIMD optimization. Even with this simple example the code grows a lot. For instance, the SIMD instructions only works on whole 16 byte vectors which leaves a rest of one to three floats to be handled separately.

Listing 5.4. Example of skeleton usage, hand SIMD optimized inner loop

```
// Definition
float addReduceVec(float *x, int n)
{
    int i;
    float res=0;
    int nVec=0;
    __align_hint(x,16,0);
    if(n>8)
    {
        nVec = n/4;
        vector float vec_res __attribute__((aligned(16)));
        vec_res = SPE_splats(0.0f);
        for(i=0;i<nVec;i++)
        {
            vec_res = SPE_add(vec_res, ((vector float*)x)[i]);
        }
        res += SPE_extract(vec_res, 0) + SPE_extract(vec_res, 1)
            + SPE_extract(vec_res, 2) + SPE_extract(vec_res, 3);
    }
    for(i=nVec*4;i<n;i++)
    {
        res+=x[i];
    }
    return res;
}
// Usage
res = sDistReduceLocalFunc(&addReduceVec, x, N);
```

5.3.4 SIMD Function Generation With Macros

To provide the user the power of SIMD optimization, without the drawbacks of doing it by hand, a simple function definition language implemented as C preprocessor macros was developed. A function defined using these macros expands to a SIMD optimized parallel function. The macro language covers a selection of standard basic and higher level math functions. It is also quite easy to expand by adding definitions to a header file. Many of the functions has a close mapping to one or a few cell SIMD instructions and some are mapped to functions in the IBM Simdmath library [9]. See Listing 5.5 for a usage example using this method.

Listing 5.5. Example of skeleton usage, macro generated approach.

```
// Definition
DEF_REDUCE_FUNC_S(my_sum, t1, BL_NONE,
    BL_SADD(t1, op1, op2))
// Usage
res = my_sum(x, N);
```

5.3.5 Performance Differences on User Provided Function Approaches

The performance difference with the different approaches for specifying user provided functions is huge. The summation example with the simple approach in Listing 5.2 is approximately 40 times slower (inner loop only) than the macro generated SIMD optimized version (Listing 5.5). A performance comparison can be seen in Figures 5.2 and 5.3. Figure 5.2 shows the performance of the whole skeleton function and Figure 5.3 shows the performance for the calculation part only. The knee in the first figure is due to the main memory bandwidth bottleneck, i.e. the SPE:s can perform the calculations faster than the operands can be transferred from main memory. The hand SIMD optimized (hand vectorized) and the macro generated functions perform identically.

5.4 Macro Skeleton Language

As described in Chapter 5.3.4 a small language for function definition was developed to enable the user access to the powerful SIMD optimizations without the drawbacks of hand SIMD optimization.

Constants are defined with the macro `BL_SCONST(name, val)` for single precision or `BL_DCONST(name, val)` for double precision. The `name` argument is the constant's name and the `val` argument is constant's value. Value can either be a numerical constant (e.g. 3.0) or a global variable. The skeleton cannot change a constant's value.

Calculation functions are defined with macros such as `BL_SADD(name, arg1, arg2)` or `BL_DADD(name, arg1, arg2)`. Here, `name` is the name of the function's result. No function results can have the same name inside a single skeleton definition (single-assignment property). `Arg1` and `arg2` are the arguments to the calculation function. Those arguments can be either the

name of a constant, the name of an other function result or one of the argument element from the argument arrays. The argument arrays are named op if there is only one argument array and op1, op2 or op3 respectively if there are two or three argument arrays. By referring from a calculation macro instance to the result names of previous calculation macros a flattened expression tree of macro instances is created. Macro naming conventions state that functions prefixed with `BL_S` are single precision and functions prefixed with `BL_D` are double precision. See appendix B for a listing of all available calculation functions.

The main part of the macro system is the code generation macro. There is one macro per combination of skeleton type, number of argument arrays and data type (i.e. single precision or double precision). The usage of a code generation macro looks like `DEF_MAP_TWO_FUNC_S(fname,res,const1 const2 ... constn, mac1 mac2 ... macn)`. The `fname` argument is the name of the function that is to be generated. The `res` argument is the name of the calculation function result that is the return value for the whole defined function. The `const` arguments are the needed constants. They are separated from the calculation functions because of performance reasons. `mac` arguments are the calculation functions. See Appendix B for a listing of all available code generation macros.

See Listing 5.6 for a demonstrative example. The resulting generated code can be seen in Listing 5.7. The generated code uses the same map function internally as the user provided inner loop version of the map (See Chapter 5.3.2).

Listing 5.6. Macro language example

```
// result= (op1 + 4) * op2
// definition
DEF_MAP_TWO_FUNC_S(map_func_name, res,
 BL_SCONST(const_4, 4.0f),
 BL_SADD(op1p4, op1, const_4)
 BL_SMUL(res, op1p4, op2))
// usage
map_func_name(block_dist_array_1, block_dist_array_2,
     block_dist_array_res, N, BL_STEP);
```

Listing 5.7. Macro generated code

```
float map_func_name_2op (float op1S, float op2S)
{
    vector float op1 __attribute__ ((aligned (16))) = SPE_splats (op1S);
    vector float op2 __attribute__ ((aligned (16))) = SPE_splats (op2S);
    vector float const_4 __attribute__ ((aligned (16))) = SPE_splats (4.0f);
    vector float op1p4 = SPE_add (op1, const_4);
    vector float res = SPE_mul (op1p4, op2);
    return SPE_extract (res, 0);
}

void map_func_name_local (float *x1, float *x2, float *res, int n)
{
    int i;
    int nVec = 0;
    if (n >= 4)
    {
        vector float const_4 __attribute__ ((aligned (16))) =
            SPE_splats (4.0f);
        nVec = n / 4;
        for (i = 0; i < nVec; i++)
        {
            vector float op1 __attribute__ ((aligned (16))) =
                ((vector float *) x1)[i];
            vector float op2 __attribute__ ((aligned (16))) =
                ((vector float *) x2)[i];
            vector float op1p4 = SPE_add (op1, const_4);
            vector float res = SPE_mul (op1p4, op2);
            ((vector float *) res)[i] = res;
        }
    }
    for (i = nVec * 4; i < n; i++)
    {
        res[i] = map_func_name_2op (x1[i], x2[i]);
    }
}
void map_func_name (BlockDistArray * x1, BlockDistArray * x2,
    BlockDistArray * res, int n, enum bl_do_step do_step)
{
    sDistMapTwoLocalFunc (&map_func_name_local, x1, x2, res, n, do_step);
}
```

5.5 Block Lib Implementation

5.5.1 Memory Management

A large part of BlockLib is memory management. Argument arrays are mostly NestStep block distributed arrays. These arrays are based in main memory and have to be transferred between main memory and the SPE:s local store. All transfers are double buffered. BlockLib contains some double buffered memory management primitives. The primitives also optimize buffer alignment making sure 128 byte alignment is used wherever possible. See Chapter 2.8.3 for details on the importance of proper alignment.

The memory management primitives are also available trough the Blocklib API to ease memory management for the library user even outside the skeleton functions. See Listing 5.8 for a usage example. The example shows a SPE reading and processing all elements of its part of a block distributed array.

Listing 5.8. Example of usage of the memory management functions

```
BArrF_Handler baX;
init_BArr_Put(&baX, x, 4096); //init handler. X is a block distributed array

NestStep_step();
{
    while(get_sw_BArr(&baX)) // get chunks. returns 0 when all chunks is read
    {
        // baX.current is current working array
        // baX.currentSize is size of current working array
        calculate(baX.current, baX.currentSize)
    }
    wait_put_BArr_done(&baRes); // wait for last put
}
NestStep_end_step();
```

5.5.2 Synchronization

The synchronization and combine functionality in the NestStep run time environment is slow as currently implemented. The functions are very generic and all communication involves the PPE. This became a major bottleneck as BlockLib was optimized. The skeleton functions does not require the full genericity of the NestStep functions. A new set of specialized non generic inter SPE synchronization and combining functions was developed to solve these performance problems. BlockLib can be compiled either with the native NestStep synchronization functions or with the BlockLib versions. This is chosen with a C define in one of the header files. There is no difference between the to versions except the performance from the library user's point of view.

The BlockLib synchronization and inter SPE communication functions are available via the BlockLib API for usage outside of the skeletons but should be used with caution as these do not conform to the BSP model.

The difference in performance for real usage can be seen the benchmarks in Chapter 6.3.4.

Signals

BlockLib synchronization is based on Cell *Signals*. Signals are sent trough special signal registers in each SPE:s MFC. Each SPE:s control area (which includes the signal registers) and local store is memory mapped and all SPE:s and the PPE can transfer data to and from every other SPE with DMA. An SPE sends signals to other SPE:s by writing to the other SPE:s signal registers. In BlockLib the signal register is used in OR-mode. This means that everything written to the register is bitwise OR-ed together instead of overwritten, so multiple SPE:s can signal one single SPE at the same time. For example, when SPE two signals SPE five it sets bit two in one of SPE five's signal register. When SPE five reads its signal register it can detect that it has received a signal from SPE two. With eight SPE:s eight bits are needed for each kind of signal. The SPE:s have two signal registers each and one of them is used by BlockLib for its three kinds of signals (barrier synchronization, message available and message acknowledge).

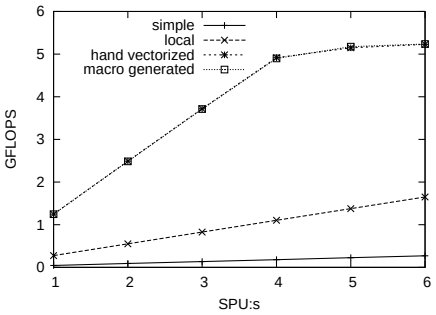

Figure 5.2. Performance of different implementations of summation (GFLOPS).

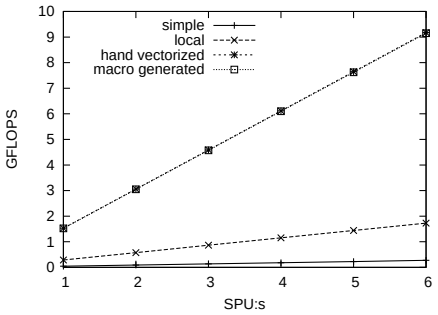

Figure 5.3. Performance of different implementations with Calculation part only (GFLOPS).

Group Barrier Synchronisation

BlockLib has a group synchronization function that uses signals. The implementation is naive but fast. One SPE is coordinator of the synchronization barrier. All SPE:s except the coordinator SPE sends a signal to the coordinator. When the coordinator has received a signal from all other SPE:s it sends a signal back to each of them. The synchronization takes less than a μs.

Message Passing

BlockLib has a set of message passing primitives. These primitives enables the user to send arrays (up to 16 kiB) between SPE:s without involving neither the PPE nor the main memory. If more than one SPE need a certain chunk of main memory one SPE can get it from main memory and then send it to the other SPE:s over the on chip interconnect buss using message passing. This will lower main memory bandwidth usage and the replication of the data does not require a full NestStep combine.

Group Synchronization with Data Combine

BlockLib has a variant of the group synchronization that also combines a data entity, such as a double or a small array (up to 2 kiB). The function is called with a value and after the synchronization all SPE:s have an array of all SPE:s values. This way all SPE:s can calculate a concurrent state for this variable (such as the maximum or the sum) with only one synchronization.

Chapter 6

Evaluation

The evaluation of BlockLib was done through synthetic benchmarks and by porting a real vector based computational application to the Cell and NestStep using the library. The synthetic benchmarks evaluate the separate skeletons and the ported application evaluates both the absolute performance relative to other computer systems and the usability of the library.

All benchmarks were run in Linux on a Playstation 3. The Syntethic benchmarks and time distribution were measured by counting processor ticks with the SPU decrementer register and the real application by measuring the total execution time with the `time` command. The processor ticks were converted to seconds with the `timebase` found in `/proc/cpuinfo`.

6.1 Time Distribution Graphs

The time distribution graphs such as in Figure 6.3 show where time is spent in the skeleton function. Time in *calculating* is time spent in the inner loop. This is how much time that is spent doing the actual useful calculations. *Dma wait* is time spent idle waiting for DMA transfers to or from the main memory. This value indicates if the memory bandwidth is a bottleneck. *Combine* is time spent either in NestStep combines or in the equivalent message passing based substitute presented in Chapter 5.5.2. This is functionality such as group synchronization and spreading of results. *Unaccounted* is everything else, such as internal copying and other administrative code. The values presented in the time distribution graphs are the average of each SPE's own measurements. For instance, the time presented as spent in *calculating* is the average of how much time each SPE spend in this part of the programs.

6.2 Synthetic Benchmarks

The skeleton functions were tested with synthetic benchmarks. The presented results are the average of 5 runs. All arrays are $5 * 1024^2$ elements long.

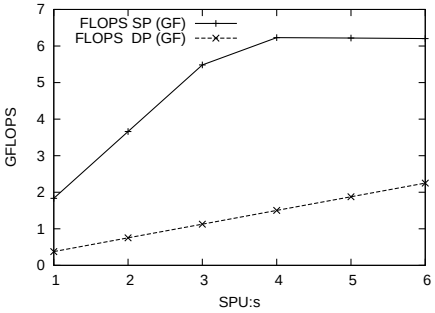

Figure 6.1. Performance of map skeleton in GFLOPS.

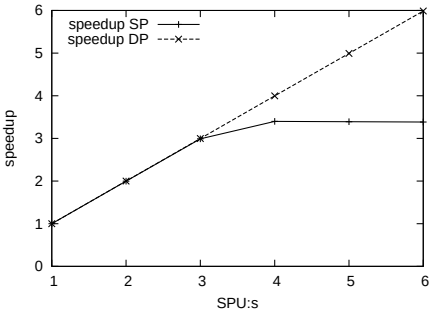

Figure 6.2. Speedup of map skeleton.

6.2.1 Performance

Map

The map skeleton was benchmarked with the function $\texttt{MAX}(op1, op2) * (op1 - op2)$ were $op1$ and $op2$ are elements from two argument arrays. This function has $4N$ float operations (as the `MAX` is one compare and one select).

The skeleton works well for double precision floats. It scales perfect even at six SPE:s. The single point version only scales well up to three SPE:s and there is no additional speedup with more than four. The reason for this can be observed in Figure 6.3 where dma wait increase for four and more SPE:s. This map skeleton transfers two arrays from, and one back, to the main memory and the four FLOP:s are too fast to compute for the main memory bus to keep up with. Single precision floats are much faster to compute than double precision floats which is why this is a bigger problem for the single precision map. More calculations for each element would improve the scaling for single precision floats. See Listing 6.1 for the relevant test code and Figures 6.1 to 6.4 for graphs.

Listing 6.1. Map function used in benchmark.

```
// Definition
// result = MAX(op1, op2) * (op1-op2)
DEF_MAP_TWO_FUNC(map_func, t3,
     BL_NONE,
     BL_MAX(t1, op1, op2)
     BL_SUB(t2, op1, op2)
     BL_MUL(t3, t1,t2))
// Usage
map_func(x, y, z, N, BL_STEP);
```

Reduce

The reduce skeleton was benchmarked with summation. Summation is $N-1$ float operations. The performance of the reduce is similar to the map performance above. The double precision version scales perfect and the single precision is limited by the memory bandwidth. This reduce only has to transfer one array

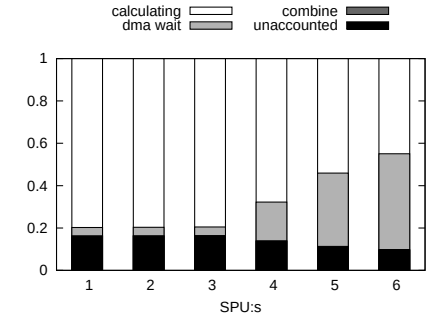

Figure 6.3. Time distribution of Map skeleton with single precision.

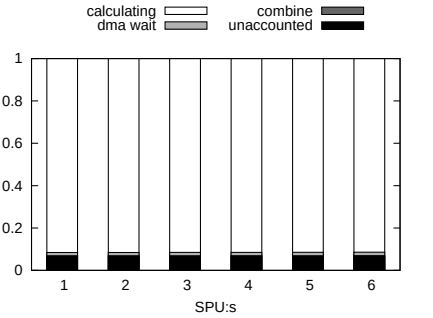

Figure 6.4. Time distribution of Map skeleton with double precision.

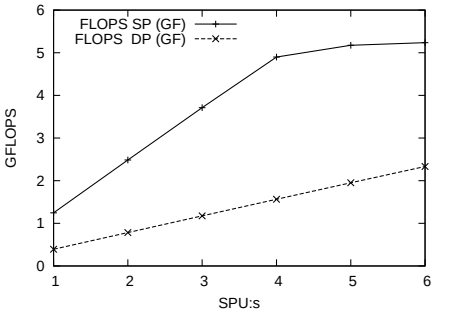

Figure 6.5. Performance of reduce skeleton in GFLOPS.

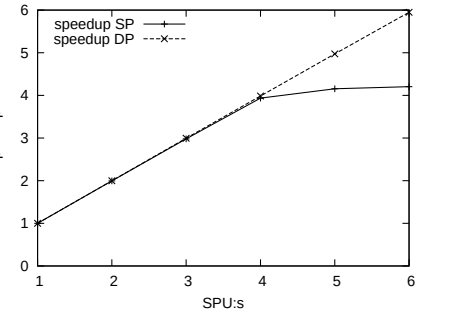

Figure 6.6. Speedup of reduce skeleton.

from main memory but has only one operation to calculate for each element. This benchmarked example runs out of main memory bandwidth at five SPE:s. See Listing 6.2 for relevant test code and Figures 6.5 to 6.8 for graphs.

Listing 6.2. Reduce function used in benchmark.

```
// Definition
DEF_REDUCE_FUNC_S(my_add_red, t1, BL_NONE,
    BL_SADD(t1, op1, op2))
// Usage
res = my_add_red(x, N);
```

Map-Reduce

The map-reduce skeleton was tested with a dot product calculation. This function has $2N - 1$ float operations. The dot product was implemented both with the map-reduce skeleton and with separate map and reduce skeletons. The two implementations are compared in Figures 6.9 to 6.14. The combined map-reduce skeleton outperforms separate map and reduce by more than a factor two with

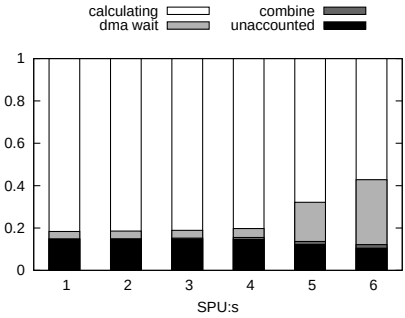

Figure 6.7. Time distribution of reduce skeleton with single precision.

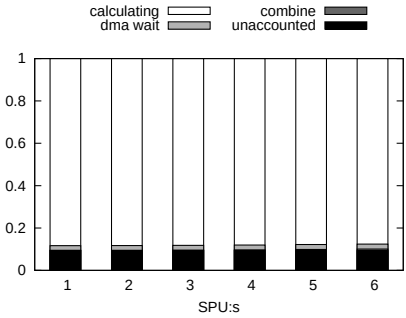

Figure 6.8. Time distribution of reduce skeleton with double precision.

more than two SPE:s. This is due to less transfers to and from main memory and less loads from and stores to the SPE local store. See Listings 6.3 and 6.4 for relevant test code.

Listing 6.3. Dot product defined with map-reduce skeleton.

```
// Definition
DEF_MAP_TWO_AND_REDUCE_FUNC_S(my_dot, mres, rres, BL_NONE,
     BL_SMUL(mres, m_op1, m_op2),
     BL_SADD(rres, r_op1, r_op2))
// Usage
res = my_dot(x,y,N);
```

Listing 6.4. Dot product defined with separate map and reduce skeletons.

```
// Definition
DEF_MAP_TWO_FUNC_S(my_map, t1, BL_NONE,
     BL_SMUL(t1, op1, op2))
DEF_REDUCE_FUNC_S(my_red,t1, BL_NONE,
     BL_SADD(t1, op1, op2))
// Usage
my_map(x,y,z,N, BL_STEP);
res= my_red(z,N);
```

Overlapped Map

The overlapped map was tested with a small convolution kernel with five coefficients. This calculation has nine float operations per element of the vector. A small kernel was chosen to put stress on the skeleton. The overlapped map is a bit slower than the normal map due to some internal copying, distribution of edge areas and alignment problems with loading operands from local store. See Listing 6.5 for test code and Figures 6.15 to 6.18 for graphs.

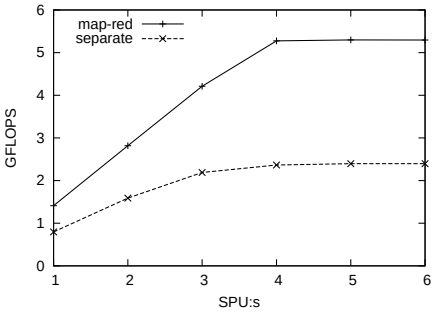

Figure 6.9. Dot product implemented with map-reduce and with separate map and reduce. Performance in GFLOPS, single precision.

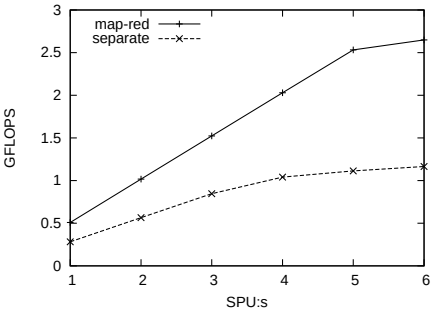

Figure 6.10. Dot product implemented with map-reduce and with separate map and reduce, Performance in GFLOPS, double precision.

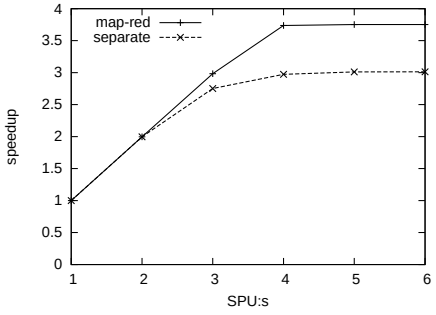

Figure 6.11. Speedup dot product implemented with map-reduce and with separate map and reduce with single precision.

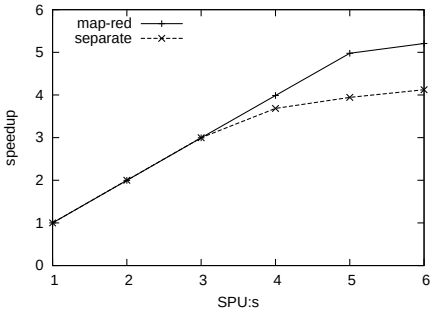

Figure 6.12. Speedup of dot product implemented with map-reduce and with separate map and reduce with, double precision.

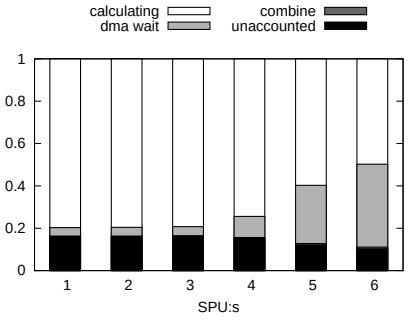

Figure 6.13. Time distribution of map-reduce skeleton with single precision.

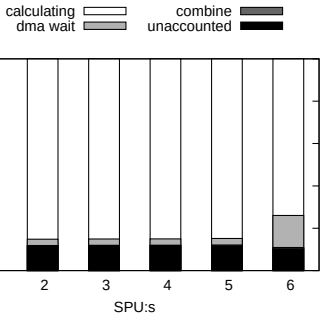

Figure 6.14. Time distribution of map-reduce skeleton with double precision.

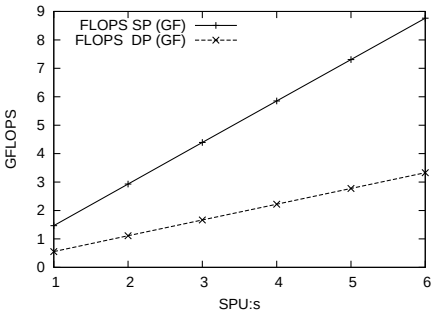

Figure 6.15. Performance of map with overlap skeleton in GFLOPS.

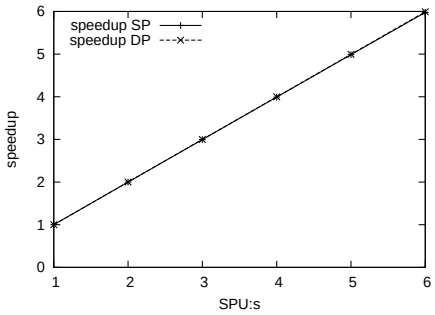

Figure 6.16. Speedup of map with overlap skeleton.

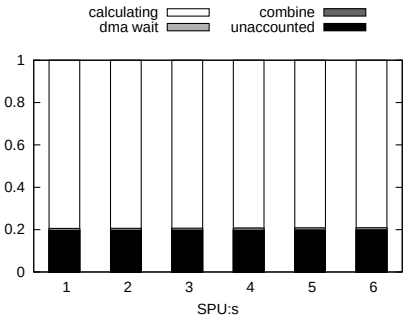

Figure 6.17. Time distribution of map with overlap skeleton with single precision.

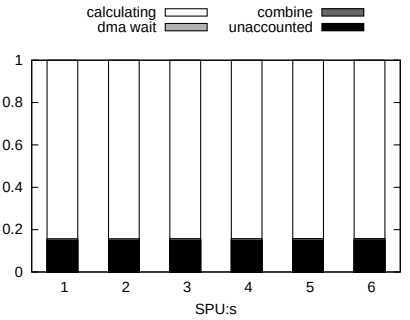

Figure 6.18. Time distribution of map with overlap skeleton with double precision.

Listing 6.5. A discrete filter implemented with the *overlapped map*.

```
// Definition
DEF_OVERLAPPED_MAP_ONE_FUNC_S(conv_s, vs5, 8,
    BL_SCONST(p0, 0.4f)
    BL_SCONST(p1, 0.2f)
    BL_SCONST(p2, 0.1f),
    BL_SINDEXED(vm2, -2)
    BL_SINDEXED(vm1, -1)
    BL_SINDEXED(v, 0)
    BL_SINDEXED(vp1, 1)
    BL_SINDEXED(vp2, 2)
    BL_SMUL(vs1, vm2, p2)
    BL_SMUL_ADD(vs2, vm1, p1, vs1)
    BL_SMUL_ADD(vs3, v, p0, vs2)
    BL_SMUL_ADD(vs4, vp1, p1, vs3)
    BL_SMUL_ADD(vs5, vp2, p2, vs4))
// Usage
conv_s(x,y,N,BL_ZERO);
```

Performance of Inner Loop

The performance shown in the benchmarks above is very far from the Cell's theoretical peak performance. This is mostly because of the few FLOPS per element.

It is hard to achieve near peak performance even when the data is in the SPE local store. Loop overhead, load and store delays, computation pipeline delay cause data dependencies to stall the SPE. This is illustrated by Figure 6.19. The graph shows just the calculation loops, excluding main memory bandwidth limitations, synchronisations delays etc. The lower curve shows the calculation of a simple sum. This calculation needs only one new operand per calculation. The upper curve shows a more advanced calculation with three arguments and a total of 24 calculations per three operands. The later also uses the combined *mull_add* operation. The 8 operations per operand and the combined instruction make this calculation reach 38.8 GFLOPS compared to the 9.2 GFLOPS of the summation. More operations per operand would increase the performance even more.

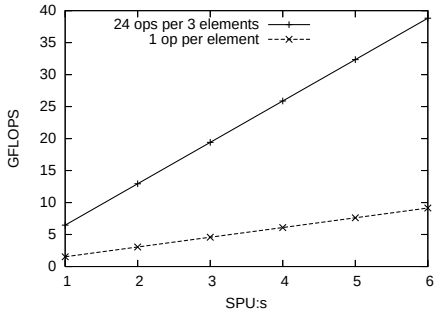

Figure 6.19. Performance of inner loops with different number of operations per element.

IBM Cell BLAS

A small benchmark was written for the IBM Cell BLAS library [6] to have something non generic but similar in ease of use as a reference point. This library looks like a normal CBLAS library and is called from a PPE program. The code using this library looks like sequential code and the library manage the SPE:s with SPE thread creation and parallelization etc. The benchmarked function is the dot product which is also used to benchmark the map-reduce skeleton above. The benchmark program was derived from one of IBM's example program for the BLAS library. See Figures 6.20 and 6.21 for performance graphs. The benchmark in Figure 6.20 uses the same sized arrays as all other benchmarks in this chapter. This graph should be compared with Figures 6.9 and 6.10 which calculates the same thing using NestStep and BlockLib. The IBM BLAS benchmark calls the CBLAS function 100 times directly after each other to suppress any library initialization overhead in the result. The IBM BLAS library is slightly faster than the BlockLib version with one SPE but does not scale well with the number of SPE:s. The speed with one SPE indicates that the BLAS library may perform and scale better with more computational heavy BLAS operations such as matrix-matrix multiplication.

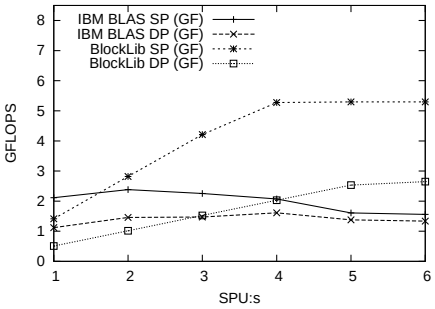

Figure 6.20. Performance of IBM Cell BLAS library, using 5M elements per array, in GFLOPS.

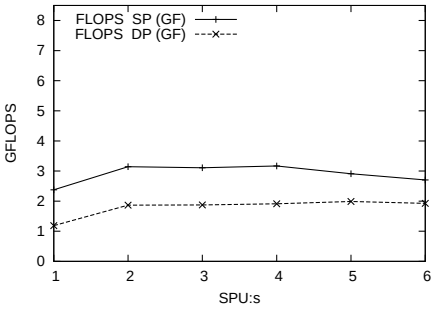

Figure 6.21. Performance of IBM Cell BLAS library, using 10M elements per array, in GFLOPS.

6.3 Real Program — ODE Solver

To evaluate the over all performance of the BlockLib in a realistic usage scenario, a real numerical computation program was ported to the Cell and NestStep using the BlockLib library.

6.3.1 LibSolve

The application of choice is LibSolve which is an Ordinary Differential Equation (*ODE*) solver developed by Matthias Korch and Thomas Rauber [16]. The library solves initial value problems for systems of first order ODE:s. Their library was chosen because of the extensive benchmarks included in their paper.

LibSolve is a big library which includes many variants of the solvers' different parts. There is sequential, shared memory parallel and distributed memory parallel solvers, a set of Runge Kutta-methods and a few ODE-problems. LibSolve links these parts together at runtime.

6.3.2 ODE problem

The ODE problems are implemented as functions for initialization of arrays and functions implementing the right side evaluation function. Here we consider ODE problems with a spatial access pattern that are created by spatial discretization of partial equations by the method of lines. The ODE problems have a property named *access distance*. The access distance is the maximum distance between argument vector elements needed in the evaluation of an ODE right hand side function and the evaluated element itself. If the access distance is large the solver has to keep a large part of the ODE system accessible at all times. In their paper [16] Korch and Rauber take advantage of small access distance to pipeline the solver. In this port we use small access distance to be able to run the evaluation function within a SPE's limited local store.

2D Brusselator

LibSolve implements two problems consisting of two dimensional Brusselator equations. Both problems are spatial discretized and result in $2N^2$ equations (ODE size) were N is the grid size. The first variant (BRUSS2D-ROW) is a more conventional discretization but has an access distance of N^2. The second variant (BRUSS2D-MIX) is re-ordered in such a way that the access distance is limited to $2N$.

6.3.3 Porting

Solver Algorithm

LibSolve implements a lot of different solver algorithms. We used the simplest vector based sequential algorithm for the port. This algorithm is a very conventional embedded Runge Kutta solver. The core of the algorithm is a solver function of just a few hundred lines of code. The code consists mainly of loops over vectors of problem size. Most of the loops have independent iterations and fit well into the map pattern. The rest of the loops also reduce the result to a maximum or minimum value, which maps directly to the map-reduce pattern. All those loops were translated into BlockLib skeleton definitions. LibSolve is based on double precision floats. The BlockLib port is therefore also based on doubles to get comparable and verifiable results.

Optimized Algorithm

The solver algorithm was later optimized to improve performance and reduce memory consumption. Several vectors could be skipped altogether by combining several loops into one. This reduced memory consumption enough to fit the largest test case into the PS3's limited main memory.

Right Hand Side Evaluation Function

The evaluation function in the port differs from the one used in LibSolve in two aspects. The original evaluation function contains a loop over all elements and a function call for each element. The version in the port does most of the calculations directly in the loop for most elements. This enables SIMD optimizations. The original evaluation function contained two things that has a devastating effect on performance on Cell SPE:s. The first is lots of if statements, the other is function calls. The SPE:s have no branch prediction making branching expensive.

6.3.4 Performance

The performance is measured with the `time` command (`/usr/bin/time` on Fedora Core). This measures the total execution time and thus includes initialization of NestStep and allocation and initialization of arrays. This is the time that matters in real usage. The measurements are from one run for each test case. All runs are long enough to give consistent results and some are too long to be run multiple

times. The results in the time distribution graph are the average values from all used SPE:s.

Test Data Sets

Four data sets were used for performance testing. These are the same that was used in [16] for the BRUSS2D-MIX problem. The grid sizes of the brusselators are $N = 250$, $N = 500$, $N = 750$ and $N = 1000$. A grid size of N gives $2N^2$ equations. This gives the data sets problem sizes of 125000, 500000, 1125000 and 2000000 equations. The larger data sets require a bigger number of steps to calculate resulting in a workload that grows faster than the number of equations.

Absolute Performance

To get a comparison of real absolute performance, our results are compared to execution times for some systems in Korch and Raubers paper [16]. Sequential execution time for their Pentium 4 workstation is 2198 seconds for the largest BRUSS2D-MIX data set (see Table 6 [16]). The Pentium 4 has a clock speed of 3.0 GHz which is similar to the PlayStation 3's clock speed of 3.2 GHz, but has very different architecture.

The compared versions differs in some aspects. They calculate the same result with the same overall algorithm but the code is compiled with different compilers and the evaluation function differs in structure. The differences make the above comparison inaccurate. To get a more accurate comparison we benchmarked a PC version of the solver that is as similar to the BlockLib port as possible. Korch and Raubers Benchmark is still interesting for comparison as that is the performance of the original software. This way we make sure to compare the BlockLib port with the fastest of the available PC versions.

Before the porting of LibSolve to BlockLib and NestStep a working minimal sequential solver for ordinary computers was extracted from LibSolve. This contained only the components that were going to be used for the BlockLib port and did not have any advanced composition system as LibSolve has. This program was then ported to BlockLib, NestStep and the Cell. To be able to measure the difference in performance between NestStep, BlockLib, the Cell and ordinary sequential C code on an ordinary PC the restructure part of the optimization of the right side evaluation function was back ported to minimal solver. The minimal solver was benchmarked with and without this optimization on a few different ordinary x86 PC machines. The benchmarks were done using the data set with grid size 1000 (2000000 equations).

The tested machines were a dual Pentium 4 Xeon 3.4GHz 64bit cluster node using GCC and ICC and a Intel Core 2 duo 2.4GHz 64bit workstation using GCC. Both machines were running Linux. As the benchmark is sequential it makes no difference that the machines are SMP systems. See Table 6.1 for the execution times. Original, optimized and SIMD in Table 6.1 refers only to the evaluation functions. The rest of code in the BlockLib port is automatically SIMD optimized in all test cases.

The BlockLib port is 4.57 times faster then the Core 2 and 5.02 times faster then the P4 Xeon. It is 7.25 times faster than the original software on the P4 workstation. The difference in speed between the BlockLib version with the optimization and the one without is huge. This SIMD optimization of the evaluation function has just a minor part in this. The optimized version takes 350 seconds with the exact same evaluation function code as the x86 optimized versions. The benchmark also illustrates how badly the SPE:s perform on if statements and function calls.

machine	original	optimized	SIMD
P4	2198 [16]	-	-
Opteron	1979 [16]	-	-
P4 Xeon, GCC	2386.47	1626.17	-
P4 Xeon, ICC	2175.52	1520.23	-
Core 2, GCC	1628.20	1386.56	-
Cell, 6 SPE:s, GCC	816.01	350.66	303.14

Table 6.1. Execution Times for the minimal sequential PC solver and the Cell port (using six SPE:s).

Synchronization Performance

Figures 6.28 and 6.29 show the same benchmark as Figures 6.22 and 6.23 but compiled with the native NestStep synchronization and combining functions. The sequential performance is almost identical but the scaling is ruined, especially for the smaller data sets.

Scaling

The speedup can be seen in Figure 6.23. The program scales well up to four SPE:s (speedup 3.9 for the largest data set, speedup 5.25 for six SPE:s). The main memory bandwidth becomes a bottleneck with five and six SPE:s, reducing speedup. This can be seen in Figures 6.24 to 6.27.

6.3.5 Usability

The port was done by converting loops into map and reduce definitions. We mostly did not need to take special care for neither the special characteristics of the Cell nor the parallelization. The biggest exception is the hand SIMD optimization of the evaluation function.

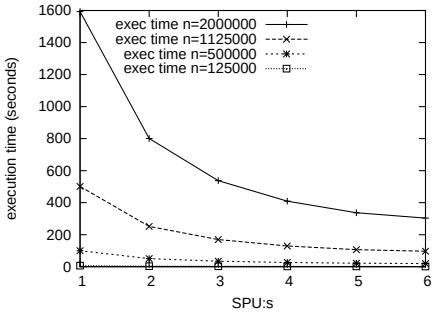

Figure 6.22. Execution times for the ode solver.

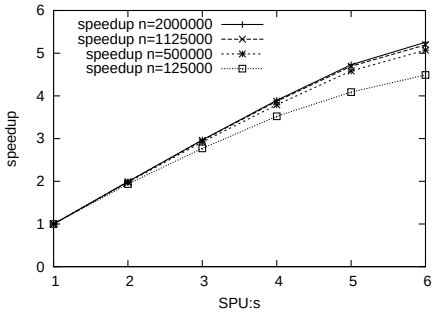

Figure 6.23. Speedup ode solver.

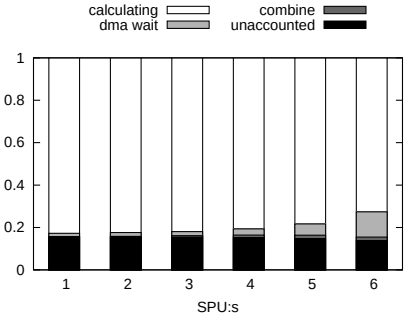

Figure 6.24. Time distribution of ODE solver with ode size 2000000.

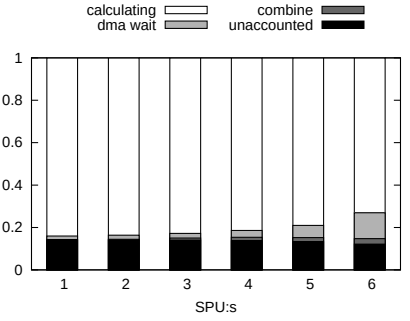

Figure 6.25. Time distribution of ODE solver with ode size 1125000.

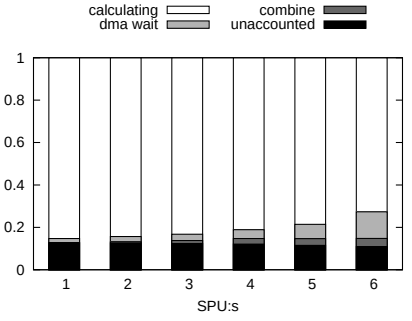

Figure 6.26. Time distribution of ODE solver with ode size 500000.

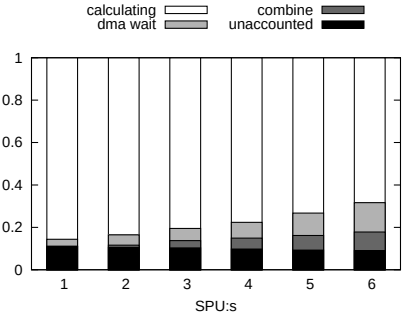

Figure 6.27. Time distribution of ODE solver with ode size 125000.

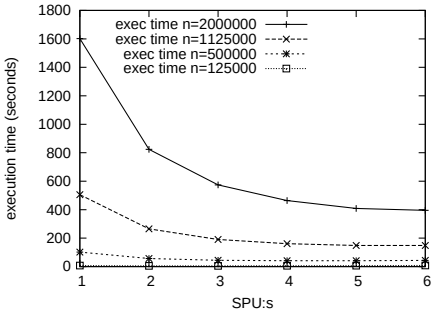

Figure 6.28. Execution times of ODE solver with NestStep native synchronisation functions.

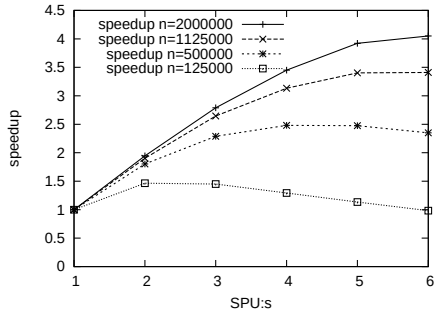

Figure 6.29. Speedup of ODE solver with NestStep native synchronisation functions.

Chapter 7

Conclusions and Future Work

The goal of this master thesis project was to make it a bit easier to program for the Cell processor in NestStep. The approach was to make it easy to perform calculations on large arrays through a set of skeletons. The evaluation chapter of this report shows that the skeletons are usable and fast enough to work for real applications.

7.1 Performance

The synthetic benchmark shows that the skeletons scale well, at least up to the limit of the memory bandwidth. The double precision versions scales almost perfect. This difference in scaling is due to that the Cell is up to 14 times slower on doubles but the doubles only consume twice the memory bandwidth. The benchmarks also show that almost all time is spent doing useful calculations, i.e. the overhead of the skeletons is low in most cases and even a hand written solution would need administrative code for memory management and such.

The automatic SIMD code generation has little or no overhead compared to hand SIMD optimized code. SIMD optimizations has proved to provide a huge performance enhancement over normal scalar C code. The generated code also perform very well when there are many calculations per operand.

The ODE solver benchmarks show that the skeletons work in a real application. The ODE solver uses double precision and confirms the synthetic tests by scaling well. Looking at the synthetic benchmarks suggests that the ODE solver would be between two and three times as fast using single precision floats.

Comparing with the results from the sequential performance on x86 systems with similar clock frequency confirms that good scaling is not just from bad real performance. It also confirms that the approach is useful for real applications.

7.2 Usability

Writing code with BlockLib is mostly like writing sequential code. The user of the library does neither need to take care of all the details of Cell programming, nor parallelization and related problems and complexity to achieve decent performance.

7.3 Future Work

7.3.1 NestStep Synchronization

The synchronization and combine functions are too slow to work with short super steps. The skeletons developed in this thesis are typically very short, mostly under 10ms even for bigger data sets. A few ms in combine in each of those have a devastating effect on overall performance and scaling. The results of the BlockLib synchronisation functions show that it is possible to do the synchronisation a lot faster. As the BlockLib synchronization functions lack the genericity of the NestStep combine they are not a drop in replacement. A future work on NestStep and Cell could look into redesigning NestStep synchronization to combine the high performance of the inter SPE signals and the genericity of NestStep combine.

7.4 Extension of BlockLib

BlockLib could be extended with more skeletons to cover more common computation patterns and the macro language could be extended to cover more small functions.

The library could be ported to some other implementation of NestStep, such as one of the PC-cluster versions.

Bibliography

[1] M. Ålind, M. Eriksson, and C. Kessler. Blocklib: A skeleton library for Cell Broadband Engine. In *Int. Workshop on Multicore Software Engineering (IWMSE-2008) at ICSE-2008, Leipzig, Germany, May 2008. ACM.*, 2008.

[2] D. A. Brokenshire. Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance. http://www-128.ibm.com/developerworks/power/library/pa-celltips1/, 2007.

[3] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman and MIT Press, 1989.

[4] S. Héman, N. Nes, M. Zukowski, and P. Boncz. Vectorized data processing on the Cell Broadband Engine. In *Proceedings of the Third International Workshop on Data Management on New Hardware (DaMoN 2007), Beijing, China June 15, 2007. ACM.*, 2007.

[5] IBM. Cell Broadband Engine architecture v1.01. In: IBM SDK for Multicore Acceleration V3.0, 2006.

[6] IBM. Basic linear algebra subprograms programmer's guide and API reference. In: IBM SDK for Multicore Acceleration V3.0, 2007.

[7] IBM. C/C++ language extensions for Cell Broadband Engine architecture. In: IBM SDK for Multicore Acceleration V3.0, 2007.

[8] IBM. Cell BE programming tutorial. In: documentation of SDK for Multicore Acceleration Version 3.0 SC33-8410-00, 2007.

[9] IBM. SIMD math library API reference manual. In: IBM SDK for Multicore Acceleration V3.0, 2007.

[10] IBM. Qs21 features. http://www-07.ibm.com/systems/id/bladecenter/qs21/features.html, January 22 2008.

[11] D. Johansson. Porting the NestStep run-time system to the CELL Broadband Engine. Master thesis LITH-IDA-EX–07/054-SE, Linköping university, 2007.

[12] J. L. T. Jörg Keller, Christoph W. Kessler. *Practical PRAM Programming*. 2000.

[13] J. Keller, C. Kessler, and J. Träff. *Practical PRAM Programming*. Wiley, New York, 2001.

[14] C. Kessler. Managing distributed shared arrays in a bulk-synchronous parallel environment. *Concurrency – Pract. Exp.*, 16:133–153, 2004.

[15] C. W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. *The J. of Supercomputing*, 17:245–262, 2000.

[16] M. Korch and T. Rauber. Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining. *J. Parallel and Distrib. Comput.*, 66:444–468, 2006.

[17] J. Sohl. A scalable run-time system for neststep on cluster supercomputers. Master thesis LITH-IDA-EX–06/011–SE, Linköping university, Sweden, 2006.

[18] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *CF '06: Proc. 3rd conf. on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

Appendix A

Glossary

A.1 Words and Abbreviations

- BLAS: Basic Linear Algebra Subprograms. A standardized interface for Linear algebra math libraries.
- BSP: Bulk Synchronous Parallel. A model for parallel programming.
- Cell BE: Cell Broadband Engine. A heterogeneous multicore processor with 1 PPE and 8 SPEs created by IBM, SONY and TOSHIBA.
- Cell SDK: Cell Software Development Kit. Development kit from IBM containing compiler, tools, documentation and example code.
- Distributed memory system: A computer system where each processor or group of processors has their own separate memory, such as a PC cluster.
- DMA: Direct Memory Access. Memory transfers that does not go trough the main processor.
- EIB: Element Interconnect Bus. The Cell's high bandwidth internal bus. Connects PPE, SPE:s, memory controller and I/O controller together.
- FLOPS: FLoating point Operations Per Second. Unit for measuring of floating point math performance.
- GCC: GNU Compiler Collection. The GNU project's compiler.
- GDB: GNU DeBugger. Debugging tool.
- ICC: Intel C++ Compiler. Highly optimized C and C++ compiler from Intel.
- MFC: Memory Flow Controller. The unit in each SPU that manage DMA-transfers.

- ODE: Ordinary Differential Equation. A kind of Differential Equations.
- PowerPC: A general purpose processor architecture by IBM.
- PPE: PowerPC Processing Element. Main processor core.
- PS3: PlayStation 3. Gaming console from SONY featuring a Cell processor.
- Shared memory system: A computer system where all processor shares a common memory.
- SIMD: Single Instruction Multiple Data
- SPE: Synergistic Processing Element. Slave processor core.
- Speedup: Relative performance, e.g. a speedup of two is twise as fast as the standard or sequantial case.
- Vector: Two or more scalars put together.

A.2 Prefixes

Prefixes used in this report.

- K Kilo, 10^3.
- Ki Kibi, 2^{10}.
- M Mega, 10^6.
- Mi Mibi, 2^{20}.
- G Giga, 10^9.
- Gi Gibi, 2^{30}.

Appendix B

BlockLib API Reference

This appendix consists of the short BlockLib API reference from the BlockLib source package.

B.1 General

```
BLock Lib API reference

This file is the short API reference to the BlockLib library. For usage examples
please see the the test program directory in blocklib/test/.


All examples in this file use floats. The double precision versions are
identical except the data type.

sDist*
    Functions prefixed with sDist are parallel distributed functions working
    with single precision (float).
dDist*
    Functions prefixed with sDist are parallel distributed functions working
    with double precision (double).

*_FUNC_S()
    Macros suffixed by _FUNC_S are code generation macros generating SIMD
    optimized parallel distributed code for single precision (float).
*_FUNC_D()
    Macros suffixed by _FUNC_S are code generation macros generating SIMD
    optimized parallel distributed code for double precision (double).

Notations
local array     - A local array is a plain continuous array in SPU local store.
                  Is and behaves like a normal c-array.
BlockDistArray  - A BlockDistArray is a NestStep distributed array.
                  See NestStep documentation for further information.


Common function arguments:
    x,x1,x2,x3  - Argument arrays.
    res         - Target array.
    N           - The number of elements in the argument arrays (all arrays have
                  to be of equal size).

Common code generation macro arguments:
    name        - The name argument is the name of the function to be generated.
    result      - The variable/calculation tree name that is the result of the
                  defined computation.
    consts      - Section for all constant definitions.
    funcs       - Section for all calculation functions/calculation trees.
```

B.2 Reduce Skeleton

```
float sDistReduce(float (*red_func)(float, float), BlockDistArray* x, int N);
double dDistReduce(double (*red_func)(double, double), BlockDistArray* x, int N);
```

```
    Reduces x to a scalar. Slow.
    red_func - Function pointer to the reduce function.
        float red_func(op1, op2)


float sDistReduceLocalFunc(float (*red_func)(float*,int),
                           BlockDistArray* x, int N);
double dDistReduceLocalFunc(double (*red_func)(double*,int),
                            BlockDistArray* x, int N);
    Reduces x to a scalar.
    red_func - Function pointer to reduce function. This function reduces a
        local array to a single value.
        float red_func(float *local_array, int N)

DEF_REDUCE_FUNC_S(name, result, consts, funcs)
DEF_REDUCE_FUNC_D(name, result, consts, funcs)
    Generates a function for reduction. The resulting function looks like:
        float name(BlockDistArray *x, int N);
```

B.3 Map Skeleton

```
Common arguments:
    do_step - Do step can either be BL_STEP or B_NO_STEP. The function forms
              a NestStep superstep if do_step i BL_STEP. If do_step is
              BL_NO_STEP the function must be part of a bigger superstep.

void sDistMapOne(float (*map_func)(float), BlockDistArray* x1,
                 BlockDistArray* res, int N, enum bl_do_step do_step);
void dDistMapOne(double (*map_func)(double), BlockDistArray* x1,
                 BlockDistArray* res, int N, enum bl_do_step do_step);
    Maps map_func onto x1 and store the result in res.
        map_func - Function pointer to map function.
        float map_func(float op)

void sDistMapOneLocalFunc(void (*map_func)(float*,float*,int), BlockDistArray* x,
                          BlockDistArray* res, int N, enum bl_do_step do_step);
void dDistMapOneLocalFunc(void (*map_func)(double*,double*,int), BlockDistArray* x,
                          BlockDistArray* res, int N, enum bl_do_step do_step);
    Maps map_func onto x and store the result in res.
        map_func - Function pointer to map function. Map a whole local array.
        void map_func(float *local_array x1, float *local_array res, int N)


void sDistMapTwo(float (*map_func)(float,float), BlockDistArray* x1,
                 BlockDistArray* x2, BlockDistArray* res, int N,
                 enum bl_do_step do_step);
void dDistMapTwo(double (*map_func)(double,double), BlockDistArray* x1,
                 BlockDistArray* x2, BlockDistArray* res, int N,
                 enum bl_do_step do_step);
    Maps map_func onto (x1,x2) and store the result in res.
        map_func - Function pointer to map function.
        float map_func(float op1, float op2)

void sDistMapTwoLocalFunc(void (*map_func)(float*,float*,float*,int),
                          BlockDistArray* x1, BlockDistArray* x2,
                          BlockDistArray* res, int N, enum bl_do_step do_step);
void dDistMapTwoLocalFunc(void (*map_func)(double*,double*,double*,int),
                          BlockDistArray* x1, BlockDistArray* x2,
                          BlockDistArray* res, int N, enum bl_do_step do_step);
    Maps map_func onto (x1,x2) and store the result in res.
        map_func - Function pointer to map function. Map a whole local array.
        void map_func(float *local_array x1, float *local_array x2,
                      float *local_array res, int N)


void sDistMapThree(float (*map_func)(float,float,float), BlockDistArray* x1,
                   BlockDistArray* x2, BlockDistArray* x3, BlockDistArray* res,
                   int N, enum bl_do_step do_step);
void dDistMapThree(double (*map_func)(double,double,double), BlockDistArray* x1,
                   BlockDistArray* x2, BlockDistArray* x3, BlockDistArray* res,
                   int N, enum bl_do_step do_step);
    Maps map_func onto (x1,x2,x3) and store the result in res.
        map_func - Function pointer to map function.
        float map_func(float op1, float op2, float op3)

void sDistMapThreeLocalFunc(void (*map_func)(float*,float*,float*,float*,int),
                            BlockDistArray* x1,BlockDistArray* x2,
                            BlockDistArray* x3, BlockDistArray* res,
                            int N, enum bl_do_step do_step);
void dDistMapThreeLocalFunc(void (*map_func)(double*,double*,double*,double*,int),
                            BlockDistArray* x1,BlockDistArray* x2,
                            BlockDistArray* x3, BlockDistArray* res,
                            int N, enum bl_do_step do_step);
    Maps map_func onto (x1,x2,x3) and store the result in res.
        map_func - Function pointer to map function. Map a whole local array.
        void map_func(float *local_array x1, float *local_array x2,
                      float *local_array x3, float *local_array res, int N)
```

```
DEF_MAP_ONE_FUNC_S(name, result, consts, funcs)
DEF_MAP_ONE_FUNC_D(name, result, consts, funcs)
    Generates a map function. The resulting function looks like:
        void name(BlockDistArray *x1, BlockDistArray *res,  int N,
                  enum  bl_do_step do_step)

DEF_MAP_TWO_FUNC_S(name, result, consts, funcs)
DEF_MAP_TWO_FUNC_D(name, result, consts, funcs)
    Generates a map function. The resulting function looks like:
        void name(BlockDistArray *x1, BlockDistArray *x2, BlockDistArray *res,
                  int N, enum  bl_do_step do_step)

DEF_MAP_THREE_FUNC_S(name, result, consts, funcs)
DEF_MAP_THREE_FUNC_D(name, result, consts, funcs)
    Generates a map function. The resulting function looks like:
        void name(BlockDistArray *x1, BlockDistArray *x2, BlockDistArray *x3,
                  BlockDistArray *res,  int N, enum  bl_do_step do_step)
```

B.4 Map-Reduce Skeleton

```
float
sDistMapOneAndReduceLocalFunc(float (*map_red_func)(float*,int),
                              float (*red_func)(float*,int),
                              BlockDistArray* x, int N);
double
dDistMapOneAndReduceLocalFunc(double (*map_red_func)(double*,int),
                              double (*red_func)(double*,int),
                              BlockDistArray* x, int N);
    Combined map and reduce. map_red_func maps and reduces each local array
    and red_func reduces all partial results (i.e. for the results of
    map_red_func).
        map_red_func - Function pointer to map+reduce function. Maps and reduce
        a whole local array.
            float map_red_func(float *local_array x1, int N)
        red_func - Function pointer to reduce function. This function reduces a
        local array to a single value.
            float red_func(float *local_array, int N)

float
sDistMapTwoAndReduceLocalFunc(float (*map_red_func)(float*,float*,int),
                              float (*red_func)(float*,int),
                              BlockDistArray* x1,BlockDistArray* x2, int N);
double
dDistMapTwoAndReduceLocalFunc(double (*map_red_func)(double*,double*,int),
                              double (*red_func)(double*,int),
                              BlockDistArray* x1,BlockDistArray* x2, int N);
    Combined map and reduce. map_red_func maps and reduces each local array
    and red_func reduces all partial results (i.e. for the results of
    map_red_func).
        map_red_func - Function pointer to map+reduce function. Maps and reduce
        a whole local array.
            float map_red_func(float *local_array x1, float *local_array x2,
                               int N)
        red_func - Function pointer to reduce function. This function reduces a
        local array to a single value.
            float red_func(float *local_array, int N)


float
sDistMapThreeAndReduceLocalFunc(float (*map_red_func)(float*,float*,float*,int),
                                float (*red_func)(float*,int),BlockDistArray* x1,
                                BlockDistArray* x2,BlockDistArray* x3, int N);
double
dDistMapThreeAndReduceLocalFunc(double (*map_red_func)(double*,double*,double*,int)
    ,
                                double (*red_func)(double*,int),BlockDistArray* x1,
                                BlockDistArray* x2,BlockDistArray* x3, int N);
    Combined map and reduce. map_red_func maps and reduces each local array
    and red_func reduces all partial results (i.e. for the results of
    map_red_func).
        map_red_func - Function pointer to map+reduce function. Maps and reduce
        a whole local array.
            float map_red_func(float *local_array x1, float *local_array x2,
                               float *local_array x2, int N)
        red_func - Function pointer to reduce function. This function reduces a
        local array to a single value.
            float red_func(float *local_array, int N)


DEF_MAP_ONE_AND_REDUCE_FUNC_S(name, map_result, red_result, bl_consts,
                              map_funcs, red_funcs)
DEF_MAP_ONE_AND_REDUCE_FUNC_D(name, map_result, red_result, bl_consts,
                              map_funcs, red_funcs)
    Generates a map-reduce function. The resulting function looks like:
        float name(BlockDistArray *x1, int N)
```

```
DEF_MAP_TWO_AND_REDUCE_FUNC_S(name, map_result, red_result, bl_consts,
                              map_funcs, red_funcs)
DEF_MAP_TWO_AND_REDUCE_FUNC_D(name, map_result, red_result, bl_consts,
                              map_funcs, red_funcs)
    Generates a map-reduce function. The resulting function looks like:
        float name(BlockDistArray *x1, BlockDistArray *x2, int N)

DEF_MAP_THREE_AND_REDUCE_FUNC_S(name, map_result, red_result, bl_consts,
                              map_funcs, red_funcs)
DEF_MAP_THREE_AND_REDUCE_FUNC_D(name, map_result, red_result, bl_consts,
                              map_funcs, red_funcs)
    Generates a map-reduce function. The resulting function looks like:
        float name(BlockDistArray *x1, BlockDistArray *x2, BlockDistArray *x3,
                   int N)
```

B.5 Overlapped Map Skeleton

```
Common arguments:
    overlap - Overlap is the needed access distance. This is limited in the
              current implementation to 4088 floats or 2044 doubles. It is also
              limited to N / number of SPU:s.
    edge    - Edge specifies the edge policy. If edge is BL_ZERO all reads
              outside array returns zero. If edge is BL_CYCLIC the array
              is cyclic.

void sDistOverlappedMap(float (*map_func)(float*, int), BlockDistArray* x,
                        BlockDistArray* res, int N, int overlap,
                        enum bl_edge_value edge);
void dDistOverlappedMap(double (*map_func)(double*, int), BlockDistArray* x,
                        BlockDistArray* res, int N, int overlap,
                        enum bl_edge_value edge);
    Maps map_func onto x and store result in res.
        map_func - Function pointer to map function. *x is a pointer to the
        argument. The skeleton guarantees that x[+-overlap] contains valid
        data. Index is the element index of x[0] in the BlockDistArray.
            float map_func(float *x, int index)


void sDistOverlappedMapLocalFunc(void (*map_func)(float*,float*,int,int),
                                 BlockDistArray* x, BlockDistArray* res, int N,
                                 int overlap, enum bl_edge_value edge);
void dDistOverlappedMapLocalFunc(void (*map_func)(double*,double*,int,int),
                                 BlockDistArray* x, BlockDistArray* res, int N,
                                 int overlap, enum bl_edge_value edge);
    Maps map_func onto x and store result in res.
        map_func - Function pointer to map function. *x is a pointer to the
        arguments. The skeleton guarantees that x[-overlap] to x[N+overlap]
        contains valid data. Index is the element index of x[0] in the
        BlockDistArray.
            void map_func(float *x, float *red, int index, int N)



BL_SINDEXED(name, offset)
BL_DINDEXED(name, offset)
    Macro for naming nearby argument elements. Name is the argument name and
    offset is the distance from the current element. This macro must be put in
    the function section of code generation macros. It can not be change but
    are not a constant.

DEF_OVERLAPPED_MAP_ONE_FUNC_S(name, result, overlap, consts, funcs)
DEF_OVERLAPPED_MAP_ONE_FUNC_D(name, result, overlap, consts, funcs)
    Generates an overlapped map function. The result looks like:
        void name(BlockDistArray *x, BlockDistArray *res,
                  int n, enum bl_edge_value edge)
```

B.6 Constants and Math Functions

```
General

BL_S*
    All macros prefixed by BL_S are for single precision (float)
BL_D*
    All macros prefixed by BL_D are for double precision (double)

Constants
BL_SCONST(name, val)
BL_DCONST(name, val)
    Defintion of a constant.
        name - the name of the constant.
        val - numerical constant (i.e. 3.4f for float, 3.4 for double)

Misc
BL_NONE
```

```
    Expand to nothing. Just for readability.


Mathematical functions
    name - The name of the result of the function. Two mathematical functions
    in the same skeleton can not have the same name.
    o1 - function argument one
    o2 - function argument two
    o3 - function argument three


Basic arithmetic:

BL_SADD(name, o1,o2): o1 + o2
BL_DADD(name, o1,o2)
BL_SSUB(name, o1,o2): o1 - o2
BL_DSUB(name, o1,o2)
BL_SMUL(name, o1,o2): o1 * o2
BL_DMUL(name, o1,o2)
BL_SMIN(name, o1,o2): o1 < o2 ? o1 : o2
BL_DMIN(name, o1,o2)
BL_SMAX(name, o1,o2): o1 > o2 ? o1 : o2
BL_DMAX(name, o1,o2)
BL_SDIV(name, o1,o2): o1 / o2
BL_DDIV(name, o1,o2)

Dual operations:

BL_SMUL_ADD(name, o1,o2,o3): o1 * o2 + o3
BL_DMUL_ADD(name, o1,o2,o3)
BL_SMUL_SUB(name, o1,o2,o3): o1 * o2 - o3
BL_DMUL_SUB(name, o1,o2,o3)


Misc functions:

BL_SSQRT(name, o1): sqrt(o1)
BL_DSQRT(name, o1)
BL_SEXP(name, o1): exp(o1)
BL_DEXP(name, o1)
BL_SLOG(name, o1): log(o1)
BL_DLOG(name, o1)
BL_SRECIP(name, o1): 1 / o1
BL_DRECIP(name, o1)
BL_SMOD(name, o1, o2): fmod(o1,o2)
BL_DMOD(name, o1, o2)
BL_SABS(name, o1): abs(o1)
BL_DABS(name, o1)


Trigonometric:

BL_SSIN(name, o1): sin(o1)
BL_DSIN(name, o1)
BL_SCOS(name, o1): cos(o1)
BL_DCOS(name, o1)
BL_SASIN(name, o1): arcsin(o1)
BL_DASIN(name, o1)
BL_SACOS(name, o1): arccos(o1)
BL_DACOS(name, o1)
BL_STAN(name, o1): tan(o1)
BL_DTAN(name, o1)
BL_SATAN(name, o1): arctan(o1)
BL_DATAN(name, o1)
```

B.7 Helper Functions

B.7.1 Block Distributed Array (BArrF/BArrD) Handlers

```
Block Distributed Array (BArrF/BArrD) Handlers

General:
All functions in this section are available in the following versions:
BArrF: Block Distributed Array, floats
BArrD: Block Distributed Array, double
PArrF: Private Array, floats
PArrD: Private Array, Dobule

void get_BArrF_dA(BlockDistArray* _barr, float* lsaddr, int lbound, int ubound);
void store_BArrF_dA(BlockDistArray* _barr, float* lsaddr, int lbound, int ubound);
void get_BArrD_dA(BlockDistArray* _barr, double* lsaddr, int lbound, int ubound);
void store_BArrD_dA(BlockDistArray* _barr, double* lsaddr, int lbound, int ubound);
    Wraper functions for NestStep array get/store functions. These functions can
    handle arbitrary alignment for the upper bound. The lower bound still has has
    to be 16byte aligned and the size still has to be <= 16kiB. The range is as
    with the NestStep versions _inclusive_ (size = ubound-lbound-1).
```

```
BArrF_Handler
BArrD_Handler
PArrF_Handler
PArrD_Handler
    Handler structures. These are slightly over 32kiB in size due to embedded
    double buffering buffers. The size can be reduced by lowering the global
    chunk sizes.
    Interesting members:
        current      - Current local array.
        currentSize  - Size of current array
        currentStart - Index of first element in current array.

void init_BArrF_Get(BArrF_Handler *h, BlockDistArray* a, int bs);
void init_BArrF_Put(BArrF_Handler *h, BlockDistArray* a, int bs);
    Initialization functions for the above handlers.
        bs - block size/chunk size. Will not lower the handlers memory
             consumption.

int get_sw_BArrF(BArrF_Handler *h);
int put_sw_BArrF(BArrF_Handler *h);
    Gets/stores chunk and get next buffer. Will start DMA of next chunk before
    returning. Return value is the current chunk size. Returns Zero when all
    chunks are done.

void wait_put_BArrF_done(BArrF_Handler *h);
    Wait for the DMA transfers from the last put to finish.
```

B.7.2 Synchronization

```
Synchronization functions using inter SPE signals. They are a specialized
alternative to the NestStep built in combines for some special cases.

void blGroupSync();
    A group synchronization using the synchronization system chosen at
    compiletime.

void blMPGroupSync();
    Group Synchronization using inter SPU signals.

void blMPDistSync(volatile void* data, void* localElement, unsigned int elementSize
    );
    Group synchronization with data synchronization. The array data will contain
    all SPU:s localElement:s after synchronization. localElement can be any data
    structure smaller than 2kiB but has to be padded to a factor of 16 byte in
    size if bigger than 8 bytes. Elements of size 8 or less must be naturally
    aligned.

Message Passing
Message Passing functions using inter SPU signals and inter SPU DMA.

void blSendMsg(volatile void * data, int size, int target);
    Send array data to SPU target. Data must be naturally aligned. Size is
    payload (data) size in bytes.

void blWaitMsgAck(int source);
    Wait for receiver to acknowledge message. This is sent by blWaitMsgDMA when
    DMA transfer is done.

int blRecvMsg(volatile void * data, int source);
    Wait for message to arrive. Start DMA of message payload when message
    arrive. Data is payload target. Returns the payload size in bytes. Payload
    Transfer is not finished when function returns.

void blWaitMsgDMA(int source);
    Wait for message payload to complete. This functions also sends message
    acknowledge.

Signals

void blSendSignal1Bit(int bit, int target);
    Send bit 'bit' as signal to target. Use with cation when used together
    with other BlockLib inter SPU synchronizations.

void blWaitSignal1Bit(int bit);
    Wait for bit 'bit' to arrive.
```

B.7.3 Pipeline

```
pipeF_Handler;
    Handler for a pipeline. This structure is 48kiB due to tripple buffering.

    Interesting members:
    current.p     - pointer to current array.
    current.size  - size of current array.
```

```
    current.start - start index of first element in current.p.


void init_pipeF_first(pipeF_Handler *h, PrivateArray* a, int toRank, int bs);
    Initialize handler. First SPU in pipeline. A is data source. Bs is chunk
    size. Bs must be a factor of 4 or of size 1 or 2. ToRank is next SPU in
    pipeline.

void init_pipeF_mid(pipeF_Handler *h, int fromRank, int toRank, int bs);
    Initialize handler. Internal SPU in pipeline. Bs is chunk
    size, must be the same as the first SPU's chunk size. FromRank is previous SPU
    in pipeline, toRank is next.

void init_pipeF_last(pipeF_Handler *h, PrivateArray* a, int fromRank, int bs);
    Initialize handler. Last SPU in pipeline. A is data source. Bs is chunk
    size, must be the same as the first SPU's chunk size. FromRank is previous
    SPU in pipeline. A is target private array. A may be NULL if pipeline
    stream is not to be saved.

int step_PipeF(pipeF_Handler *h);
    Step pipeline one chunk forward. The return value is the size of the the
    current chunk. The return value is zero when pipeline is finished.
```

Appendix C

Code for Test Programs

C.1 Map Test Code

```
/* ******************************************
 * BlockLib Test Program - Map
 * ******************************************/

#include <stdio.h>
#include <NestStep_spulib/neststep_spu.h>
#include "block_lib.h"
#include <spu_intrinsics.h>
// Initialization value for the counter
#define DEC_INIT_VAL 0XFFFFFFFF


#define MAX(a,b) (a>b ? a:b)

int N = 1024*1024*5;

int MY_RANK;

BlockDistArray* x;
BlockDistArray* y;
BlockDistArray* z;

float func(float left, float right)
{
    return MAX(left,right) *(left-right);
}

void func_local(float *x1, float *x2, float *res, int N)
{
    int i;
    for(i=0;i<N;i++)
        res[i] = MAX(x1[i],x2[i]) * (x1[i]- x2[i]);
}

float sqrt_func(float x)
{
    return sqrt(x);
}

float gen(int i)
{
    return (float)i;
}

float genMod(int i)
{
    return (float)(i%34);
}


DEF_GENERATE_FUNC_S(mGen, iF, BL_NONE, BL_NONE)

DEF_MAP_TWO_FUNC_S(map_func, t3, BL_NONE,
BL_SMAX(t1, op1, op2)
BL_SSUB(t2, op1, op2)
BL_SMUL(t3, t1,t2)
```

```
)

float map_saxpy_alpha = 3;
DEF_MAP_TWO_FUNC_S(map_saxpy, t1,
BL_SCONST(alpha, map_saxpy_alpha),
BL_SMUL_ADD(t1, op1, alpha, op2)
)

DEF_MAP_ONE_FUNC_S(sqrt_one, t1, BL_NONE,
BL_SSQRT(t1, op)
)

DEF_MAP_THREE_FUNC_S(map_matr, rq,
BL_SCONST(c11, 1.0f)
BL_SCONST(c12, 2.0f)
BL_SCONST(c13, 3.0f)
BL_SCONST(c14, 4.0f)
BL_SCONST(c21, 4.0f)
BL_SCONST(c22, 5.0f)
BL_SCONST(c23, 6.0f)
BL_SCONST(c24, 7.0f)
BL_SCONST(c31, 7.0f)
BL_SCONST(c32, 8.0f)
BL_SCONST(c33, 9.0f)
BL_SCONST(c34, 10.0f),

BL_SMUL_ADD(x11, op1, c11, c14)
BL_SMUL_ADD(x12, op1, c12, x11)
BL_SMUL_ADD(x1r, op1, c13, x12)
BL_SMUL(x1q, x1r, x1r)

BL_SMUL_ADD(x21, op2, c21, c24)
BL_SMUL_ADD(x22, op2, c22, x21)
BL_SMUL_ADD(x2r, op2, c23, x22)
BL_SMUL_ADD(x2q, x2r, x2r, x1q)

BL_SMUL_ADD(x31, op3, c31, c34)
BL_SMUL_ADD(x32, op3, c32, x31)
BL_SMUL_ADD(x3r, op3, c33, x32)
BL_SMUL_ADD(rq, x3r, x3r, x2q)
)


void calculate(){

    sDistGenerate(&gen, x, N,BL_STEP);
    sDistGenerate(&genMod, y, N, BL_STEP);
    sDistGenerate(&gen, z, N, BL_STEP);

    clearTimers(&bl_time);

    sDistGenerate(&gen, x, N,BL_STEP);

    printBArrF(x,N, 65);


    prs0("\nSimpel generator (1 ops/n)")
    prs0("opn: 1")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);


    mGen(x, N,BL_STEP);

    printBArrF(x,N, 65);

    prs0("\nMacro generated generator (1 ops/n)")
    prs0("opn: 1")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    map_matr(x, y,z, z, N, BL_STEP);

    prs0("\nmap lenght in quadr after matrix transformation, macro generated (23
        ops/n)")
    prs0("opn: 23")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    sDistMapTwo(&func, x, y, z, N, BL_STEP);

    //printBArrF(z,N);
```

```
    prs0("\nmap max(a,b)*(a-b), simple (4 ops/n)")
    prs0("opn: 4")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    sDistMapTwoLocalFunc(&func_local,x, y, z, N, BL_STEP);

    prs0("map max(a,b)*(a-b), local map function (4 ops/n)")
    prs0("opn: 4")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    map_func(x, y, z, N, BL_STEP);

    prs0("\nmap max(a,b)*(a-b), macro generated (4 ops/n)")
    prs0("opn: 4")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    sDistMapOne(&sqrt_func, x,z,N, BL_STEP);

    printBArrF(z,N, 65);

    prs0("\nmap one sqrt simple (1 sqrt/n)")
    prs0("opn: 1")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    sqrt_one(x,z,N,BL_STEP);

    prs0("\nmap one sqrt macro generated (1 sqrt/n)")
    prs0("opn: 1")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    map_saxpy_alpha = 3.5;
    map_saxpy(x,y,z,N, BL_STEP);

    printBArrF(z,N, 65);

    prs0("\nsaxpy macro generated (2 ops/n)")
    prs0("opn: 2")
    printTimersDetailed_flops(bl_time, N);
    prs0("")
    clearTimers(&bl_time);

}

int main(unsigned long long speid, Addr64 argp, Addr64 envp)
{
    spu_write_decrementer(DEC_INIT_VAL);
    int i;

    NestStep_SPU_init(argp);
    MY_RANK = NestStep_get_rank();
    Name name;
    name.procedure = 1;
    name.relative = 0;

    x = new_BlockDistArray(name,FVAR,N);
    name.relative++;
    y = new_BlockDistArray(name,FVAR,N);
    name.relative++;
    z = new_BlockDistArray(name,FVAR,N);


    for(i=0;i<2;i++){
        if(MY_RANK==0)
            printf("\n\niteration %d:\n", i);
        calculate();
    }

    prd0(N)

    NestStep_SPU_finalize();
```

```
    return 0;
}
```

C.2 Reduce Test Code

```
/* ******************************************
 * BlockLib Test Program - Reduce
 * ******************************************/

#include <stdio.h>
#include <NestStep_spulib/neststep_spu.h>
#include "block_lib.h"
#include <spu_intrinsics.h>

// Initialization value for the counter
#define DEC_INIT_VAL 0XFFFFFFFF

int N = 1024*1024*5;

int MY_RANK;

BlockDistArray* x;

float add(float left, float right)
{
    return left+right;
}

float addReduce(float *x, int n)
{
    int i;
    float res=x[0];
    for(i=1;i<n;i++)
    {
        res+=x[i];
    }
    return res;
}

float addReduceVec(float *x, int n)
{
    int i;
    float res=0;
    int nVec=0;
    __align__hint(x,16,0);
    if(n>8)
    {
        nVec = n/4;
        vector float vec_res __attribute__((aligned(16)));
        vec_res = spu_splats(0.0f);
        for(i=0;i<nVec;i++)
        {
            vec_res = spu_add(vec_res, ((vector float*)x)[i]);
        }
        res += spu_extract(vec_res, 0) + spu_extract(vec_res, 1) + spu_extract(
            vec_res, 2) + spu_extract(vec_res, 3);
    }
    for(i=nVec*4;i<n;i++)
    {
        res+=x[i];
    }
    return res;
}


float gen(int i)
{
    return (float)i;
}

DEF_REDUCE_FUNC_S(max, t1, BL_NONE,
BL_SMAX(t1, op1, op2)
)

DEF_REDUCE_FUNC_S(my_add_red, t1, BL_NONE,
BL_SADD(t1, op1, op2)
)

void calculate(){
    int i, j, no_chunks;
    i = j = 0;
    float res;

    sDistGenerate(&gen, x, N, BL_STEP);

    clearTimers(&bl_time);
    res = sDistReduce(&add, x, N);
```

```
    prs0("\nSimple sum (1 ops/n)")
    prs0("opn: 1")
    prf0(res)
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    res = sDistReduceLocalFunc(&addReduce, x, N);

    prs0("\nLocal sum (1 ops/n)")
    prs0("opn: 1")
    prf0(res)
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    res = sDistReduceLocalFunc(&addReduceVec, x, N);

    prs0("\nLocal vectorized sum (1 ops/n)")
    prs0("opn: 1")
    prf0(res)
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    res = my_add_red(x, N);

    prs0("\nLocal macro generated sum (1 ops/n)")
    prs0("opn: 1")
    prf0(res)
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);;

    res = max(x, N);

    prs0("\nLocal macro generated max (2 ops/n)")
    prs0("opn: 2")
    prf0(res)
    printTimersDetailed_flops(bl_time, N);
    prs0("")
    clearTimers(&bl_time);;
}

int main(unsigned long long speid, Addr64 argp, Addr64 envp)
{
    spu_write_decrementer(DEC_INIT_VAL);
    int i;


    NestStep_SPU_init(argp);
    MY_RANK = NestStep_get_rank();
    Name name;
    name.procedure = 1;
    name.relative = 0;

    x = new_BlockDistArray(name,FVAR,N);
    for(i=0;i<2;i++){
        if(MY_RANK==0)
            printf("\n\niteration %d:\n", i);
        calculate();
    }

    prd0(N)

    NestStep_SPU_finalize();

    return 0;
}
```

C.3 Map-Reduce Test Code

```
/* ******************************************
 * BlockLib Test Program - Map-Reduce
 * ******************************************/

#include <stdio.h>
#include <NestStep_spulib/neststep_spu.h>
#include "block_lib.h"
```

```
#include <spu_intrinsics.h>
// Initialization value for the counter
#define DEC_INIT_VAL 0XFFFFFFFF

int N = 1024*1024*5;

int MY_RANK;

BlockDistArray* x;
BlockDistArray* y;
BlockDistArray* z;

float gen(int i)
{
    return (float)i;
}

DEF_MAP_THREE_AND_REDUCE_FUNC_S(matr_len, rq, t1,
BL_SCONST(c11, 1.0f)
BL_SCONST(c12, 2.0f)
BL_SCONST(c13, 3.0f)
BL_SCONST(c14, 4.0f)
BL_SCONST(c21, 4.0f)
BL_SCONST(c22, 5.0f)
BL_SCONST(c23, 6.0f)
BL_SCONST(c24, 7.0f)
BL_SCONST(c31, 7.0f)
BL_SCONST(c32, 8.0f)
BL_SCONST(c33, 9.0f)
BL_SCONST(c34, 10.0f),

BL_SMUL_ADD(x11, m_op1, c11, c14)
BL_SMUL_ADD(x12, m_op1, c12, x11)
BL_SMUL_ADD(x1r, m_op1, c13, x12)
BL_SMUL(x1q, x1r, x1r)

BL_SMUL_ADD(x21, m_op2, c21, c24)
BL_SMUL_ADD(x22, m_op2, c22, x21)
BL_SMUL_ADD(x2r, m_op2, c23, x22)
BL_SMUL_ADD(x2q, x2r, x2r, x1q)

BL_SMUL_ADD(x31, m_op3, c31, c34)
BL_SMUL_ADD(x32, m_op3, c32, x31)
BL_SMUL_ADD(x3r, m_op3, c33, x32)
BL_SMUL_ADD(rq, x3r, x3r, x2q),
BL_SMAX(t1,r_op1,r_op2)
)

DEF_MAP_TWO_FUNC_S(my_map, t1, BL_NONE,
BL_SMUL(t1, op1, op2)
)

DEF_REDUCE_FUNC_S(my_red,t1, BL_NONE,
BL_SADD(t1, op1, op2)
)

DEF_MAP_TWO_AND_REDUCE_FUNC_S(my_dot, mres, rres, BL_NONE, BL_SMUL(mres, m_op1,
     m_op2), BL_SADD(rres, r_op1, r_op2))

void calculate(){
    float res;
    sDistGenerate(&gen, x, N, BL_STEP);
    sDistGenerate(&gen, y, N, BL_STEP);
    sDistGenerate(&gen, z, N, BL_STEP);

    blGroupSync();
    clearTimers(&bl_time);

    res = my_dot(x,y,N);

    prs0("\nmap-reduce dot (2 ops/n)")
    prs0("opn: 2")
    prf0(res)
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    my_map(x,y,z,N, BL_STEP);
    res= my_red(z,N);

    prs0("\nseperate map and reduce dot (2 ops/n)")
    prs0("opn: 2")
    prf0(res)
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);
```

```
    res = matr_len(x,y,z,N);

    prs0("lenght␣of␣vector␣longest␣after␣matrix␣transformation␣(24␣ops/n)\n")
    prs0("opn:␣24")
    prf0(sqrt(res))
    printTimersDetailed_flops(bl_time, N);
    prs0("")
}

int main(unsigned long long speid, Addr64 argp, Addr64 envp)
{
    spu_write_decrementer(DEC_INIT_VAL);
    int i;

    NestStep_SPU_init(argp);
    MY_RANK = NestStep_get_rank();
    Name name;
    name.procedure = 1;
    name.relative = 0;

    x = new_BlockDistArray(name,FVAR,N);
    name.relative++;
    y = new_BlockDistArray(name,FVAR,N);
    name.relative++;
    z = new_BlockDistArray(name,FVAR,N);


    for(i=0;i<2;i++){
        if(MY_RANK==0)
            printf("\n\niteration␣%d:\n", i);
        calculate();
    }

    prd0(N)
    NestStep_SPU_finalize();

    return 0;
}
```

C.4 Overlapped Map Test Code

```
/* ********************************************
 * BlockLib Test Program - Overlapped Map
 * ********************************************/


#include <stdio.h>
#include <NestStep_spulib/neststep_spu.h>
#include "block_lib.h"
#include <spu_intrinsics.h>
#define DEC_INIT_VAL 0XFFFFFFFF


int N = 1024*1024*5;

int MY_RANK;

BlockDistArray* x;
BlockDistArray* xD;
BlockDistArray* y;
BlockDistArray* yD;


float gen(int i)
{
    return (float)i;
}

double gen_d(int i)
{
    return (double)i;
}

float genMod(int i)
{
    return (float)(i%34);
}


float overlapTest(float *x, int i)
{
    return x[-2]*0.1 + x[-1]*0.2 + x[0]*0.4 + x[1]*0.2 + x[2]*0.1;
}

void overlapLocal(float *x, float *res, int startN, int n)
{
    int i;
    for(i=0;i<n;i++)
```

```
        res[i] = x[i-2]*0.1 + x[i-1]*0.2 + x[i]*0.4 + x[i+1]*0.2 + x[i+2]*0.1;
}

double dOverlapTest(double *x, int i)
{
    return x[-2]*0.1 + x[-1]*0.2 + x[0]*0.4 + x[1]*0.2 + x[2]*0.1;
}

void dOverlapLocal(double *x, double *res, int startN, int n)
{
    int i;
    for(i=0;i<n;i++)
        res[i] = x[i-2]*0.1 + x[i-1]*0.2 + x[i]*0.4 + x[i+1]*0.2 + x[i+2]*0.1;
}

DEF_OVERLAPPED_MAP_ONE_FUNC_S(conv_s, vs5, 8,
    BL_SCONST(p0, 0.4f)
    BL_SCONST(p1, 0.2f)
    BL_SCONST(p2, 0.1f),
    BL_SINDEXED(vm2, -2)
    BL_SINDEXED(vm1, -1)
    BL_SINDEXED(v, 0)
    BL_SINDEXED(vp1, 1)
    BL_SINDEXED(vp2, 2)
    BL_SMUL(vs1, vm2, p2)
    BL_SMUL_ADD(vs2, vm1, p1, vs1)
    BL_SMUL_ADD(vs3, v, p0, vs2)
    BL_SMUL_ADD(vs4, vp1, p1, vs3)
    BL_SMUL_ADD(vs5, vp2, p2, vs4)
)

DEF_OVERLAPPED_MAP_ONE_FUNC_D(conv_d, vs5, 8,
    BL_DCONST(p0, 0.4)
    BL_DCONST(p1, 0.2)
    BL_DCONST(p2, 0.1),
    BL_DINDEXED(vm2, -2)
    BL_DINDEXED(vm1, -1)
    BL_DINDEXED(v, 0)
    BL_DINDEXED(vp1, 1)
    BL_DINDEXED(vp2, 2)
    BL_DMUL(vs1, vm2, p2)
    BL_DMUL_ADD(vs2, vm1, p1, vs1)
    BL_DMUL_ADD(vs3, v, p0, vs2)
    BL_DMUL_ADD(vs4, vp1, p1, vs3)
    BL_DMUL_ADD(vs5, vp2, p2, vs4)
)


DEF_GENERATE_FUNC_S(mGen, iF, BL_NONE, BL_NONE)
DEF_GENERATE_FUNC_D(mGen_d, iF, BL_NONE, BL_NONE)


void calculate(){
    int i, j, no_chunks;
    i = j = 0;

    mGen(x, N,BL_STEP);
    mGen(y, N,BL_STEP);
    mGen_d(xD, N, BL_STEP);
    mGen_d(yD, N, BL_STEP);


    if(MY_RANK == 0)
        printf("<<<<>>>>\n\n");

    clearTimers(&bl_time);

    sDistOverlappedMap(&overlapTest, x, y, N, 4, BL_ZERO);
//  sDistOverlappedMap(&overlapTest, x, y, N, 4, BL_CYCLIC);

    printBArrF(y,N, 65);

    prs0("\nSimple overlapped (single) (9 ops/n)")
    prs0("opn: 9")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    sDistOverlappedMapLocalFunc(&overlapLocal, x, y, N, 4, BL_ZERO);
//  sDistOverlappedMapLocalFunc(&overlapLocal, x, y, N, 4, BL_CYCLIC);

    printBArrF(y,N, 65);

    prs0("\nLocal overlapped (single) (9 ops/n)")
    prs0("opn: 9")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
```

```
    clearTimers(&bl_time);

    conv_s(x,y,N,BL_ZERO);
//  conv_s(x,y,N,BL_CYCLIC);

    printBArrF(y,N, 65);

    prs0("\nMacro genterated overlapped (single) (9 ops/n)")
    prs0("opn: 9")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    dDistOverlappedMap(&dOverlapTest, xD, yD, N, 4, BL_ZERO);
//  dDistOverlappedMap(&dOverlapTest, xD, yD, N, 4, BL_CYCLIC);

    printBArrD(yD,N, 65);

    prs0("\nSimple overlapped(double) (9 ops/n)")
    prs0("opn: 9")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    dDistOverlappedMapLocalFunc(&dOverlapLocal, xD, yD, N, 4, BL_ZERO);
//  dDistOverlappedMapLocalFunc(&dOverlapLocal, xD, yD, N, 4, BL_CYCLIC);

    printBArrD(yD,N, 65);

    prs0("\nLocal overlapped (double) (9 ops/n)")
    prs0("opn: 9")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);

    conv_d(xD,yD,N,BL_ZERO);
//  conv_d(xD,yD,N,BL_CYCLIC);

    printBArrD(yD,N, 65);

    prs0("\nMacro genterated overlapped (double) (9 ops/n)")
    prs0("opn: 9")
    printTimersDetailed_flops(bl_time, N);
    prs0("")

    blGroupSync();
    clearTimers(&bl_time);
}

int main(unsigned long long speid, Addr64 argp, Addr64 envp)
{
    spu_write_decrementer(DEC_INIT_VAL);
    int i;

    NestStep_SPU_init(argp);
    MY_RANK = NestStep_get_rank();
    Name name;
    name.procedure = 1;
    name.relative = 0;

    x = new_BlockDistArray(name,FVAR,N);
    name.relative++;
    xD = new_BlockDistArray(name,DVAR,N);
    name.relative++;
    y = new_BlockDistArray(name,FVAR,N);
    name.relative++;
    yD = new_BlockDistArray(name,DVAR,N);

    for(i=0;i<2;i++){
        if(MY_RANK==0)
            printf("\n\niteration %d:\n", i);
        calculate();
    }

    NestStep_SPU_finalize();

    return 0;
}
```

C.5 Pipe Test Code

```
/* ********************************************
```

```
 * BlockLib Test Program - Pipeline
 * ********************************************/

#include <stdio.h>
#include <NestStep_spulib/neststep_spu.h>
#include "block_lib.h"
#include <spu_intrinsics.h>

// Initialization value for the counter
#define DEC_INIT_VAL 0XFFFFFFFF


int rank;
int groupSize;
spu_timers timers={{0,0},{0,0},{0,0},{0,0},{0,0}};

void binPrint(unsigned int val)
{
    int u;
    char out[33];
    for(u=31;u>=0;u--)
        out[31-u]=(val&(1<<u) ? '1':'0');
    out[32]='\0';
    printf("(%d): %s\n", NestStep_get_rank(), out);
}

DEF_GENERATE_FUNC_PRIVATE_S(gen, iF, BL_NONE, BL_NONE)
DEF_GENERATE_FUNC_PRIVATE_S(genZero, zero, BL_SCONST(zero, 0.0f), BL_NONE)

int N = 1024*1024*1;

#define CHSIZE 4096

DEF_GENERATE_FUNC_S(mGen, iF, BL_NONE, BL_NONE)

int main(unsigned long long speid, Addr64 argp, Addr64 envp)
{

    spu_write_decrementer(DEC_INIT_VAL);
    int i;

    NestStep_SPU_init(argp);
    rank = NestStep_get_rank();
    groupSize = NestStep_get_size();

    Name name;
    name.procedure = 1;
    name.relative = 0;
    PrivateArray* parr;

    if(rank == 0 || rank == groupSize -1)
    {
        parr = new_PrivateArray(name, FVAR, N, (Addr64)NULL);
    }
    name.relative++;

    BlockDistArray* x;
    x = new_BlockDistArray(name,FVAR,N*groupSize);
    mGen(x, N,BL_STEP);


    if(rank == 0)
        gen(parr, N, BL_NO_STEP);
    else if(rank == groupSize -1)
        genZero(parr, N, BL_NO_STEP);

    initMp();

    clearTimers(&bl_time);
    clearTimers(&timers);
    NestStep_step();
    {
        pipeF_Handler pipe; // pipe handler
        BArr_Handler baX; // block dist array handler
        for(i=0;i<1000;i++)
        {
            startTimer(&timers.total);

            // set the pipe up.
            if(rank == 0)
                init_pipeF_first(&pipe, parr, rank+1, 4096);
            else if(rank != groupSize -1)
                init_pipeF_mid(&pipe, rank-1, rank+1, 4096);
            else
                init_pipeF_last(&pipe, NULL, rank-1, 4096);

            // init block dist array handler
            init_BArr_Get(&baX, x, 4096);
```

```
            while(step_PipeF(&pipe)) // loop trough all chunks
            {
                get_sw_BArr(&baX);
                // work on chunks here
            }
            stopTimer(&timers.total);
        }
        NestStep_combine(NULL,NULL);
    }
    NestStep_end_step();
    prs("done");


    printTimersDetailed_flops(timers,N*1000ll*4);
    printTimersDetailed(bl_time);

    if(rank == 0 || rank == groupSize -1)
        free_PrivateArray(parr);
    free_BlockDistArray(x);

    NestStep_SPU_finalize();

    return 0;
}
```

C.6 Ode Solver (Core Functions)

```
// >>>>>>>>>>>>>>>>>>>>
// Skeleton defintions

DEF_MAP_TWO_AND_REDUCE_FUNC_D(dDistSubAbsMax, abs, max, BL_NONE, BL_DSUB(sub, m_op1
    , m_op2) BL_DABS(abs, sub), BL_DMAX(max, r_op1, r_op2));
DEF_MAP_TWO_AND_REDUCE_FUNC_D(dDistMaxAbsQuot, abs, max, BL_NONE, BL_DDIV(div,
    m_op1, m_op2) BL_DABS(abs, div), BL_DMAX(max, r_op1, r_op2));

double yscalH;
DEF_MAP_TWO_FUNC_D(dDistYscal, s2, BL_DCONST(c1, 1.0e-30) BL_DCONST(h, yscalH),
BL_DABS(yabs, op1)
BL_DMUL(hstg, op2, h)
BL_DABS(hstgabs, hstg)
BL_DADD(s1, yabs, hstgabs)

//abs(h*help1 / (abs(y) + abs(h*stagev))+1.0e-30)
//(h*help) / ((fabs(y[i]) + fabs(h * stagevec[0][i])) + 1.0e-30);

DEF_MAP_THREE_AND_REDUCE_FUNC_D(dDistMaxAbsQuotYscal, abs, max, BL_DCONST(c1, 1.0e
    -30) BL_DCONST(h, yscalH),
BL_DABS(yabs, m_op1)
BL_DMUL(hstg, m_op2, h)
BL_DABS(hstgabs, hstg)
BL_DADD(s1, yabs, hstgabs)
BL_DADD(s2, s1, c1)

BL_DMUL(h_help, h, m_op3)

BL_DDIV(div, h_help, s2)
BL_DABS(abs, div),

BL_DMAX(max, r_op1, r_op2))


DEF_MAP_ONE_FUNC_D(dDistCopy, op, BL_NONE, BL_NONE)
DEF_MAP_ONE_AND_REDUCE_FUNC_D(dDistAbsMax, abs, max, BL_NONE, BL_DABS(abs, m_op1),
    BL_DMAX(max, r_op1, r_op2));
DEF_REDUCE_FUNC_D(dDistMax, max, BL_NONE, BL_DMAX(max, op1, op2))
DEF_REDUCE_FUNC_D(dDistSum, sum, BL_NONE, BL_DADD(sum, op1, op2))
DEF_REDUCE_FUNC_D(dDistMin, min, BL_NONE, BL_DMIN(min, op1, op2))
DEF_GENERATE_FUNC_D(dDistZero, zero, BL_DCONST(zero, 0.0), BL_NONE)


double map_dscale_alpha = 0;
DEF_MAP_ONE_FUNC_D(dMapScale, t1,
BL_DCONST(alpha, map_dscale_alpha),
BL_DMUL(t1, op, alpha))
void dDistScale(double alpha, BlockDistArray *x1, BlockDistArray *res, int n,
    enum bl_do_step do_step)
{
    map_dscale_alpha = alpha;
    dMapScale(x1, res, n, do_step);
}


double map_mad_two_a1 = 0;
double map_mad_two_a2 = 0;
DEF_MAP_TWO_FUNC_D(dMapMadd2, s1,
BL_DCONST(a1, map_mad_two_a1)
BL_DCONST(a2, map_mad_two_a2),
BL_DMUL(r1, a1, op1)
```

```
BL_DMUL_ADD(s1, op2, a2, r1))

void dMaddTwo(double a1, BlockDistArray *x1, double a2, BlockDistArray *x2,
     BlockDistArray *res, int n, enum bl_do_step do_step)
{
    map_mad_two_a1 = a1;
    map_mad_two_a2 = a2;
    dMapMadd2(x1, x2, res, n, do_step);
}

double map_mad_three_a1 = 0;
double map_mad_three_a2 = 0;
double map_mad_three_a3 = 0;
DEF_MAP_THREE_FUNC_D(dMapMadd3, s2,
BL_DCONST(a1, map_mad_three_a1)
BL_DCONST(a2, map_mad_three_a2)
BL_DCONST(a3, map_mad_three_a3),
BL_DMUL(r1, a1, op1)
BL_DMUL_ADD(s1, op2, a2, r1)
BL_DMUL_ADD(s2, op3, a3, s1))

void dMaddThree(double a1, BlockDistArray *x1, double a2, BlockDistArray *x2,
     double a3, BlockDistArray *x3, BlockDistArray *res, int n, enum bl_do_step
     do_step)
{
    map_mad_three_a1 = a1;
    map_mad_three_a2 = a2;
    map_mad_three_a3 = a3;
    dMapMadd3(x1, x2, x3, res, n, do_step);
}

double map_daxpy_alpha = 0;
DEF_MAP_TWO_FUNC_D(dMapDAXPY, t1,
BL_DCONST(alpha, map_daxpy_alpha),
BL_DMUL_ADD(t1, op1, alpha, op2))

void dDistAXPY(double alpha, BlockDistArray *x, BlockDistArray *y, BlockDistArray
    *res, int n, enum bl_do_step do_step)
{
    map_daxpy_alpha = alpha;
    dMapDAXPY(x,y,res,n,do_step);
}



// The code below is derived from Matthias Korchs and Thomas Raubers work LibSolve

// >>>>>>>>>>>>>>>>>>>>
// Runge Kutta Core loop

// simple port. Most of the commented code is the code before convertion to skeleton instance

void par_simplD(double t0, BlockDistArray *y0, double bf, double H,
                        BlockDistArray *y, double **a, double *b,
                        double *bs, double *c, uint s, uint ord)
{
    uint i, j, l;
    if(NestStep_get_rank() == 0)
        printf("par_simplD\n");
    // locked at seven stages, order 5;
    BlockDistArray *stagevec[7];
    // double **stagevec, *old_y, *help, *help1, *err_vector, *yscal
    BlockDistArray *old_y, *help, *help1, *err_vector, *yscal;
    double temp, error_max;
    double h, old_h, t = t0, te = t0 + H;
    double *bbs;

    prd0(ode_size);
    bbs = MALLOC(s, double);
    for (i = 0; i < s; ++i)
        bbs[i] = b[i] - bs[i];

    Name name;
    name.procedure = 5;
    name.relative = 0;

    // set up y directly ad y0 to save some ram
    y0=y;
    // moved over stage-vec init to keep ram usage down
    h = par_initial_step(ord, t0, H, y0);
    prf0(h)

    //ALLOC2D(stagevec, s, ode_size, double);
    for(i=0;i<s;i++)
    {
        stagevec[i] = new_BlockDistArray(name,DVAR,ode_size);
        name.relative++;
    }

    //  old_y = MALLOC(ode_size, double);
    //  help = MALLOC(ode_size, double);
```

```
//   help1 = MALLOC(ode_size, double);
//   err_vector = MALLOC(ode_size, double);
//   yscal = MALLOC(ode_size, double);

old_y = new_BlockDistArray(name,DVAR,ode_size);
name.relative++;
help = new_BlockDistArray(name,DVAR,ode_size);
name.relative++;
help1 = new_BlockDistArray(name,DVAR,ode_size);
name.relative++;
err_vector = new_BlockDistArray(name,DVAR,ode_size);
name.relative++;
yscal = new_BlockDistArray(name,DVAR,ode_size);
name.relative++;

// for (i = 0; i < ode_size; ++i)
// y[i] = y0[i];
//dDistCopy(y0,y,ode_size, BL_STEP);
//   printBArrD(y, ode_size, 2048);

//h = par_initial_step(ord, t0, H, y0);

int steps = 0;
while (t < te)
{
     steps++;
     for (i = 0; i < s; ++i)
     {
          //for (j = 0; j < ode_size; ++j)
          //help1[j] = 0.0;
          dDistZero(help1, ode_size, BL_STEP);

          //for (j = 0; j < i; ++j)
          //   for (l = 0; l < ode_size; ++l)
          //      help1[l] += a[i][j] * stagevec[j][l];

          for (j = 0; j < i; ++j)
               dDistAXPY(a[i][j], stagevec[j], help1, help1, ode_size, BL_STEP);

          //for (j = 0; j < ode_size; ++j)
          //   help1[j] = h * help1[j] + y[j];

          dDistAXPY(h, help1, y, help1, ode_size, BL_STEP);

          startTimer(&timer.calc);
          //ode_eval_all(t + c[i] * h, help1, stagevec[i]);
          par_eval_all(t + c[i]*h, help1, stagevec[i]);
          stopTimer(&timer.calc);
     }

     //for (i = 0; i < ode_size; ++i)
     //{
     //   help[i] = 0.0;
     //   help1[i] = 0.0;
     //}

     dDistZero(help, ode_size, BL_STEP);
     dDistZero(help1, ode_size, BL_STEP);

     for (i = 0; i < s; ++i)
     {
          //for (j = 0; j < ode_size; ++j)
          //{
               //help[j] += bbs[i] * stagevec[i][j];
               //help1[j] += b[i] * stagevec[i][j];
          //}
          dDistAXPY(bbs[i], stagevec[i], help, help, ode_size, BL_STEP);
          dDistAXPY(b[i], stagevec[i], help1, help1, ode_size, BL_STEP);
     }

     //for (i = 0; i < ode_size; ++i)
     //{
     //   old_y[i] = y[i];
     //   yscal[i] = (fabs(y[i]) + fabs(h * stagevec[0][i])) + 1.0e-30;
     //}

     dDistCopy(y,old_y, ode_size, BL_STEP);
     yscalH = h;
     dDistYscal(y, stagevec[0], yscal, ode_size, BL_STEP);

     //for (i = 0; i < ode_size; ++i)
     //{
     //   y[i] += h * help1[i];
     //   err_vector[i] = h * help[i];
     //}
     dDistAXPY(h, help1, y, y, ode_size, BL_STEP);
     dDistScale(h, help, err_vector, ode_size, BL_STEP);

     old_h = h;

     // error_max = 0.0;
```

```
        //for (i = 0; i < ode_size; ++i)
        //{
        //  temp = fabs(err_vector[i] / yscal[i]);
        //  if (temp > error_max)
        //       error_max = temp;
        //}
        error_max = dDistMaxAbsQuot(err_vector, yscal, ode_size);

        error_max = error_max / bf;
        pre0(h)
        pre0(error_max)
        if (error_max <= 1.0)        /* accept */
        {
            h *=
            MAX(1.0 / 3.0, 0.9 * pow(1.0 / error_max, 1.0 / (ord + 1.0)), double);

            t += old_h;
            //ode_update(t, y); points to a dummy for the bruss-problems
        }
        else                         /* reject */
        {
            h = MAX(0.1 * h, 0.9 * h * pow(1.0 / error_max, 1.0 / ord), double);

            //for (i = 0; i < ode_size; ++i)
            //  y[i] = old_y[i];
            dDistCopy(old_y, y, ode_size, BL_STEP);
        }

        h = MIN(h, te - t, double);
    }

    prd0(steps)
    double sum = dDistSum(y, ode_size);
    prf0(sum);


    // FREE2D(stagevec);
    // FREE(yscal);
    // FREE(old_y);
    // free(err_vector);
    // FREE(help);
    // FREE(help1);


    for(i=0;i<s;i++)
        free_BlockDistArray(stagevec[i]);

    free_BlockDistArray(old_y);
    free_BlockDistArray(help);
    free_BlockDistArray(help1);
    free_BlockDistArray(err_vector );
    free_BlockDistArray(yscal);

    FREE(bbs);
}


// Optimized version. Multiple maps in a single superstep, Multiple argument arrays to a
     single map etc
void par_simplD_opt1(double t0, BlockDistArray *y0, double bf, double H,
                     BlockDistArray *y, double **a, double *b,
                     double *bs, double *c, uint s, uint ord)
{
  uint i, j, l;

    if(NestStep_get_rank() == 0)
        printf("par_simplD_opt1\n");
    // locked at seven stages, order 5;
    BlockDistArray *stagevec[7];
    // double **stagevec, *old_y, *help, *help1, *err_vector, *yscal
    BlockDistArray *old_y, *help, *help1, *err_vector, *yscal;
    double temp, error_max;
    double h, old_h, t = t0, te = t0 + H;
    double *bbs;

    prd0(ode_size);
    bbs = MALLOC(s, double);
    for (i = 0; i < s; ++i)
        bbs[i] = b[i] - bs[i];

    Name name;
    name.procedure = 5;
    name.relative = 0;

    // set up y directly ad y0 to save some ram
    //dDistCopy(y0,y,ode_size, BL_STEP);
    y0=y;

    // moved over stage-vec init to keep ram usage down
    h = par_initial_step(ord, t0, H, y0);
```

```
prf0(h)
//ALLOC2D(stagevec, s, ode_size, double);
for(i=0;i<s;i++)
{
    stagevec[i] = new_BlockDistArray(name,DVAR,ode_size);
    name.relative++;
}

//  old_y = MALLOC(ode_size, double);
//  help = MALLOC(ode_size, double);
//  help1 = MALLOC(ode_size, double);
//  err_vector = MALLOC(ode_size, double);
//  yscal = MALLOC(ode_size, double);

help1 = new_BlockDistArray(name,DVAR,ode_size);
name.relative++;

// for (i = 0; i < ode_size; ++i)
// y[i] = y0[i];
// dDistCopy(y0,y,ode_size, BL_STEP);
// printBArrD(y, ode_size, 2048);

// h = par_initial_step(ord, t0, H, y0);

int steps = 0;
while (t < te)
{
    steps++;
    startTimer(&timer.calc);
    par_eval_all(t + c[0]*h, y, stagevec[0]);
    stopTimer(&timer.calc);
    for (i = 1; i < s; ++i)
    {
        //for (j = 0; j < ode_size; ++j)
        //help1[j] = 0.0;
        NestStep_step();
        {
            //for (j = 0; j < i; ++j)
            //  for (l = 0; l < ode_size; ++l)
            //    help1[l] += a[i][j] * stagevec[j][l];
            switch(i)
            {
                case 1:
                    dMaddTwo(a[i][0]*h, stagevec[0], 1, y, help1, ode_size,
                        BL_NO_STEP);
                    break;
                case 2:
                    dMaddThree(a[i][0]*h, stagevec[0], a[i][1]*h, stagevec[1],
                        1, y, help1, ode_size, BL_NO_STEP);
                    break;
                case 3:
                    dMaddThree(a[i][0]*h, stagevec[0], a[i][1]*h, stagevec[1],
                        a[i][2]*h, stagevec[2], help1, ode_size, BL_NO_STEP);
                    dMaddTwo(1, help1, 1, y, help1, ode_size, BL_NO_STEP);
                    break;
                case 4:
                    dMaddThree(a[i][0]*h, stagevec[0], a[i][1]*h, stagevec[1],
                        a[i][2]*h, stagevec[2], help1, ode_size, BL_NO_STEP);
                    dMaddThree(a[i][3]*h, stagevec[3], 1, help1, 1, y, help1,
                        ode_size, BL_NO_STEP);
                    break;
                case 5:
                    dMaddThree(a[i][0]*h, stagevec[0], a[i][1]*h, stagevec[1],
                        a[i][2]*h, stagevec[2], help1, ode_size, BL_NO_STEP);
                    dMaddThree(a[i][3]*h, stagevec[3], a[i][4]*h, stagevec[4],
                        1, help1, help1, ode_size, BL_NO_STEP);
                    dMaddTwo(1, help1, 1, y, help1, ode_size, BL_NO_STEP);
                    break;
                case 6:
                    dMaddThree(a[i][0]*h, stagevec[0], a[i][1]*h, stagevec[1],
                        a[i][2]*h, stagevec[2], help1, ode_size, BL_NO_STEP);
                    dMaddThree(a[i][3]*h, stagevec[3], a[i][4]*h, stagevec[4],
                        1, help1, help1, ode_size, BL_NO_STEP);
                    dMaddThree(a[i][5]*h, stagevec[5], 1, help1, 1, y, help1,
                        ode_size, BL_NO_STEP);
                    break;
            }
            startTimer(&timer.comb);
            //NestStep_combine(NULL,NULL);
            blMPGroupSync();
            stopTimer(&timer.comb);
        }
        NestStep_end_step();
        startTimer(&timer.calc);
        //ode_eval_all(t + c[i] * h, help1, stagevec[i]);
        par_eval_all(t + c[i]*h, help1, stagevec[i]);
        stopTimer(&timer.calc);
    }

    //for (i = 0; i < ode_size; ++i)
    //{
```

```
//   help[i] = 0.0;
//   help1[i] = 0.0;
//}

// calc error
NestStep_step();
{
     dMaddThree(bbs[0], stagevec[0], bbs[1], stagevec[1], bbs[2], stagevec
          [2], help1, ode_size, BL_NO_STEP);
     dMaddThree(bbs[3], stagevec[3], bbs[4], stagevec[4], 1, help1, help1,
          ode_size, BL_NO_STEP);
     dMaddThree(bbs[5], stagevec[5], bbs[6], stagevec[6], 1, help1, help1,
          ode_size, BL_NO_STEP);

     startTimer(&timer.comb);
     //NestStep_combine(NULL,NULL);
     blMPGroupSync();
     stopTimer(&timer.comb);
}
NestStep_end_step();

old_h = h;

// error_max = 0.0;
// for (i = 0; i < ode_size; ++i)
// {
//  temp = fabs(err_vector[i] / yscal[i]);
//   if (temp > error_max)
//        error_max = temp;
// }
// error_max = dDistMaxAbsQuot(err_vector, yscal, ode_size);
yscalH = h;
error_max = dDistMaxAbsQuotYscal(y, stagevec[0], help1, ode_size);
error_max = error_max / bf;



if (error_max <= 1.0)         // accept
{
     NestStep_step();
     {
          dDistZero(help1, ode_size, BL_NO_STEP);

          for (i = 0; i < s; ++i)
          {
               //for (j = 0; j < ode_size; ++j)
               //{
                    //help[j] += bbs[i] * stagevec[i][j];
                    //help1[j] += b[i] * stagevec[i][j];
               //}

               dDistAXPY(b[i], stagevec[i], help1, help1, ode_size, BL_NO_STEP
                    );
          }

          //for (i = 0; i < ode_size; ++i)
          //{
          //   old_y[i] = y[i];
          //   yscal[i] = (fabs(y[i]) + fabs(h * stagevec[0][i])) + 1.0e-30;
          //}


          //for (i = 0; i < ode_size; ++i)
          //{
          //   y[i] += h * help1[i];
          //   err_vector[i] = h * help[i];
          //}
          dDistAXPY(h, help1, y, y, ode_size, BL_NO_STEP);
          startTimer(&timer.comb);
          //NestStep_combine(NULL,NULL);
          blMPGroupSync();
          stopTimer(&timer.comb);
     }
     NestStep_end_step();

  h *=
    MAX(1.0 / 3.0, 0.9 * pow(1.0 / error_max, 1.0 / (ord + 1.0)), double);


  t += old_h;

  //ode_update(t, y); points to a dummy for the bruss-problems
}
else                          // reject
{
  h = MAX(0.1 * h, 0.9 * h * pow(1.0 / error_max, 1.0 / ord), double);

  //for (i = 0; i < ode_size; ++i)
  //   y[i] = old_y[i];
    //dDistCopy(old_y, y, ode_size, BL_STEP);
}
```

```
        h = MIN(h, te - t, double);
    }

    prd0(steps)
    double sum = dDistSum(y, ode_size);
    prf0(sum);


    // FREE2D(stagevec);
    // FREE(yscal);
    // FREE(old_y);
    // free(err_vector);
    // FREE(help);
    // FREE(help1);


    for(i=0;i<s;i++)
        free_BlockDistArray(stagevec[i]);

    free_BlockDistArray(help1);

    FREE(bbs);
}



// >>>>>>>>>>>>>>>>>>>>
// Evaluation functions

// original evaluation function structure
double brus_mix_eval_t;
void brus_mix_eval_local(double *y, double *f, int start, int n)
{
    int i;
    for(i=0;i<n;i++)
        f[i] = bruss_mix_eval_comp(start+i, brus_mix_eval_t, &y[-start]);
}


// restructured evaluation function
void brus_mix_eval_local2(double *y, double *f, int start, int n)
{
    int i;
    int u;
    int N = bruss_grid_size;
    int N2 = N+N;
    for(i=0;i<n;i++)
    {
        int k=(i+start)/(N2);
        int j=(i+start)-k*(N2);
        j >>= 1;
        if(!((k == 0) || (k == N - 1) || (j == N - 1) || (j == 0) ))
        {
            int next = (k+1)*(N2)-start-2;
            next = (next < n ? next : n);

            double alpha = bruss_alpha;
            double N1 = (double) N - 1.0;
            double ann = alpha*N1*N1;
            if((i+start)&1)
            {
                f[i] =  3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] + ann * (y[i -
                    N2] + y[i + N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
                i++;
            }
            for(;i<next-1;i+=2)
            {
                f[i] =  1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i]  + ann * (y[i -
                    N2] + y[i + N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
                i++;
                f[i] = 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] + ann *
                                (y[i - N2] + y[i + N2] + y[i - 2] + y[i + 2] - 4.0
                                    * y[i]);
                i--;
            }
            if(i<next)
            {
                if((i+start)&1)
                    f[i] =  3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] + ann * (y[
                        i - N2] + y[i + N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
                else
                    f[i] =  1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] +  ann * (y[
                        i - N2] + y[i + N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
            }
            else
                i--;
        }
        else
            f[i] = bruss_mix_eval_comp(start+i, brus_mix_eval_t, &y[-start]);
    }
```

```
}

// SIMD optimized evaluation function
void brus_mix_eval_local3(double *y, double *f, int start, int n)
{
    int i;
    int u;
    int N = bruss_grid_size;
    int N2 = N+N;
    for(i=0;i<n;i++)
    {
        int k=(i+start)/(N2);
        int j=(i+start)-k*(N2);
        j >>= 1;
        if(!((k == 0) || (k == N - 1) || (j == N - 1) || (j == 0) ))
        {
            int next = (k+1)*(N2)-start-2;
            next = (next < n ? next : n);

            double alpha = bruss_alpha;
            double N1 = (double) N - 1.0;
            vector double v_a = spu_splats(alpha);
            vector double v_N1 = spu_splats(N1);
            vector double v_4 = spu_splats(4.0);
            vector double v_3_4 = spu_splats(3.4);
            vector double v_4_4 = spu_splats(4.4);
            vector double v_1 = spu_splats(1.0);
            vector double v_ann = spu_splats(alpha*N1*N1);
            if((i+start)&1)
            {
                f[i] =  3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] + alpha *
                    N1 * N1 * (y[i - N2] + y[i + N2] + y[i - 2] + y[i + 2] -
                    4.0 * y[i]);
                i++;
            }

            if(i<next-3)
            {
                for(;i<next-3;i+=4)
                {
                    vector double ev_i = {y[i],y[i+2]};
                    vector double ev_ip1 = {y[i+1],y[i+2+1]};
                    vector double ev_ip2 = {y[i+2],y[i+2+2]};
                    vector double ev_im2 = {y[i-2],y[i+2-2]};
                    vector double ev_imN2 = {y[i-N2],y[i+2-N2]};
                    vector double ev_ipN2 = {y[i+N2],y[i+2+N2]};

                    vector double eres = spu_mul(v_ann, spu_sub(spu_add(ev_imN2,
                        spu_add(ev_ipN2, spu_add(ev_im2, ev_ip2))), spu_mul(v_4,
                        ev_i)));

                    eres = spu_add(spu_sub(spu_add(v_1, spu_mul(spu_mul(ev_i,ev_ip1
                        ), ev_i)), spu_mul(v_4_4, ev_i)),eres);

                    i++;
                    vector double ov_i = {y[i],y[i+2]};
                    vector double ov_im1 = {y[i-1],y[i+2-1]};
                    vector double ov_ip2 = {y[i+2],y[i+2+2]};
                    vector double ov_im2 = {y[i-2],y[i+2-2]};
                    vector double ov_imN2 = {y[i-N2],y[i+2-N2]};
                    vector double ov_ipN2 = {y[i+N2],y[i+2+N2]};
                    i--;
                    vector double ores = spu_mul(v_ann, spu_sub(spu_add(ov_imN2,
                        spu_add(ov_ipN2, spu_add(ov_im2, ov_ip2))),spu_mul(v_4,
                        ov_i)));
                    ores = spu_add(spu_sub(spu_mul(v_3_4, ov_im1), spu_mul(ov_im1,
                        spu_mul(ov_im1, ov_i))), ores);

                    f[i] = spu_extract(eres, 0);
                    f[i+1] = spu_extract(ores, 0);
                    f[i+2] = spu_extract(eres, 1);
                    f[i+3] = spu_extract(ores, 1);
                }
            }
            if(i<next-1)
            {
                f[i] =  1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i]  + alpha * N1 *
                    N1 * (y[i - N2] + y[i + N2] + y[i - 2] + y[i + 2] - 4.0 * y[i
                    ]);
                i++;
                f[i] = 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i]+ alpha * N1 * N1
                     * (y[i - N2] + y[i + N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
                i++;
            }
            if(i<next)
            {
                if((i+start)&1)
                    f[i] =  3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] + alpha *
                        N1 * N1 * (y[i - N2] + y[i + N2] + y[i - 2] + y[i + 2] -
                        4.0 * y[i]);
                else
```

```
                    f[i] =  1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] +  alpha *
                        N1 * N1 * (y[i - N2] + y[i + N2] + y[i - 2] + y[i + 2] -
                        4.0 * y[i]);
            }
            else
                i--;
        }
        else
            f[i] = bruss_mix_eval_comp(start+i, brus_mix_eval_t, &y[-start]);
    }
}

// skeleton callers
void dist_bruss_mix_eval_all(double t0, BlockDistArray *y , BlockDistArray *f)
{
    brus_mix_eval_t = t0;
    // dDistOverlapedMapLocalFunc(&brus_mix_eval_local, y, f, ode_size, (int)ode_acc_dist(),
        BL_ZERO);
    // dDistOverlapedMapLocalFunc(&brus_mix_eval_local2, y, f, ode_size, (int)ode_acc_dist(),
        BL_ZERO);
    dDistOverlapedMapLocalFunc(&brus_mix_eval_local3, y, f, ode_size, (int)
        ode_acc_dist()+4, BL_ZERO);
}


// evaluation of a single component
double bruss_mix_eval_comp(uint i, double t, const double *y)
{
  double alpha = bruss_alpha;
  uint N = bruss_grid_size;
  double N1 = (double) N - 1.0;
  uint N2 = N + N;
  uint k = i / N2;
  //uint j  = i % N2;
  //uint v  = j % 2;
  //j /= 2;
  uint j = i - k * N2;
  uint v = i & 1;
  j >>= 1;

  if (!v)    // --- U ------------------------------------------------------------
  {
    if (k == 0)
    {
      if (j == 0)
    // U(0,0)
    return 1.0
      + y[i] * y[i] * y[i + 1] - 4.4 * y[i]
      + alpha * N1 * N1 * (2.0 * y[i + N2] + 2.0 * y[i + 2] - 4.0 * y[i]);

      if (j == N - 1)
    // U(0,N-1)
    return 1.0
      + y[i] * y[i] * y[i + 1] - 4.4 * y[i]
      + alpha * N1 * N1 * (2.0 * y[i + N2] + 2.0 * y[i - 2] - 4.0 * y[i]);

      // U(0, j)
      return 1.0
    + y[i] * y[i] * y[i + 1] - 4.4 * y[i]
    + alpha * N1 * N1
    * (2.0 * y[i + N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
    }
    else if (k == N - 1)
    {
      if (j == 0)
    // U(N-1,0)
    return 1.0
      + y[i] * y[i] * y[i + 1] - 4.4 * y[i]
      + alpha * N1 * N1 * (2.0 * y[i - N2] + 2.0 * y[i + 2] - 4.0 * y[i]);

      if (j == N - 1)
    // U(N-1,N-1)
    return 1.0
      + y[i] * y[i] * y[i + 1] - 4.4 * y[i]
      + alpha * N1 * N1 * (2.0 * y[i - N2] + 2.0 * y[i - 2] - 4.0 * y[i]);

      // U(N-1,j)
      return 1.0
    + y[i] * y[i] * y[i + 1] - 4.4 * y[i]
    + alpha * N1 * N1
    * (2.0 * y[i - N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
    }
    else
    {
      if (j == 0)
    // U(k,0)
    return 1.0
      + y[i] * y[i] * y[i + 1] - 4.4 * y[i]
      + alpha * N1 * N1
      * (y[i - N2] + y[i + N2] + 2.0 * y[i + 2] - 4.0 * y[i]);
```

```
      if (j == N - 1)
    // U(k,N-1)
    return 1.0
      + y[i] * y[i] * y[i + 1] - 4.4 * y[i]
      + alpha * N1 * N1
      * (y[i - N2] + y[i + N2] + 2.0 * y[i - 2] - 4.0 * y[i]);

      // U(k,j) allgemein
      return 1.0
    + y[i] * y[i] * y[i + 1] - 4.4 * y[i]
    + alpha * N1 * N1
    * (y[i - N2] + y[i + N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
    }
  }
  else       // --- V ---------------------------------------------------
  {
    if (k == 0)
    {
      if (j == 0)
    // V(0,0)
    return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i]
      + alpha * N1 * N1 * (2.0 * y[i + N2] + 2.0 * y[i + 2] - 4.0 * y[i]);

      if (j == N - 1)
    // V(0,N-1)
    return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i]
      + alpha * N1 * N1 * (2.0 * y[i + N2] + 2.0 * y[i - 2] - 4.0 * y[i]);

      // V(0, j)
      return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i]
    + alpha * N1 * N1
    * (2.0 * y[i + N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
    }
    else if (k == N - 1)
    {
      if (j == 0)
    // V(N-1,0)
    return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i]
      + alpha * N1 * N1 * (2.0 * y[i - N2] + 2.0 * y[i + 2] - 4.0 * y[i]);

      if (j == N - 1)
    // V(N-1,N-1)
    return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i]
      + alpha * N1 * N1 * (2.0 * y[i - N2] + 2.0 * y[i - 2] - 4.0 * y[i]);

      // V(N-1,j)
      return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i]
    + alpha * N1 * N1
    * (2.0 * y[i - N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
    }
    else
    {
      if (j == 0)
    // V(k,0)
    return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i]
      + alpha * N1 * N1
      * (y[i - N2] + y[i + N2] + 2.0 * y[i + 2] - 4.0 * y[i]);

      if (j == N - 1)
    // V(k,N-1)
    return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i]
      + alpha * N1 * N1
      * (y[i - N2] + y[i + N2] + 2.0 * y[i - 2] - 4.0 * y[i]);
      // V(k,j) allgemein
      return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i]
    + alpha * N1 * N1
    * (y[i - N2] + y[i + N2] + y[i - 2] + y[i + 2] - 4.0 * y[i]);
    }
  }
}
```