

Scalability and Productivity of Parallel Programming Models for Heterogeneous-ISA Multi-Core Architectures with Local Memories

Roger Ferrer^{1,2}, Pieter Bellens^{1,2}, Konstantinos Koukos³, Michail Alvanos³,
Jae-Seung Yeom⁴, Scott Schneider⁴, Vicenç Beltran¹, Marc González^{1,2}, Xavier Martorell^{1,2},
Rosa M. Badia¹, Dimitrios S. Nikolopoulos³, Angelos Bilas³, Eduard Ayguadé^{1,2}

¹Barcelona Supercomputing Center
Nexus II – Jordi Girona, 29 – Barcelona, Spain

²Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya.
Edifici C6 – Campus Nord – Barcelona, Spain

³FORTH-ICS
100 N. Plastira Av. Vassilia Vouton, Heraklion, Greece

⁴Department of Computer Science
Virginia Tech
Blacksburg, VA, 24060, USA

Abstract

With the advent of multicore architectures, especially with the heterogeneous ones, programming is becoming a hard task. ...

Keywords: (D1.3) Concurrent Programming, (D2.6a) Environments for multiple-processor systems, (C.0.b) Hardware/software interfaces, (C.1.3.f) Heterogeneous (hybrid) systems

1 Introduction

Heterogeneous multi-core processors are being commoditized by major processor vendors including IBM (Cell [8]), Intel (Larrabee [30]), AMD (Fusion [1]), and Nvidia (Fermi [22, 23]). These processors provide high compute density and memory bandwidth at the chip scale. They are often regarded as effective computational accelerators and they have served this purpose well so far, as exemplified by the use of Cell processors on the first Petaflop-capable supercomputer (LANL's RoadRunner) [2].

Developing programming models and runtime environments for heterogeneous multi-core processors is a challenge and a point of extensive debate in the parallel programming community. Conventional parallel programming models assume processor homogeneity. Adapting these models to heterogeneous architectures is challenging, since the programming environment (compiler, runtime system, OS) needs to manage different core architectures and instruction sets simultaneously. Furthermore, the programming environment needs to efficiently support data transfers between heterogeneous cores, by exploiting the bandwidth of the processor, while taming latency.

Numerous industry and academic efforts explore programming models for heterogeneous multi-core processors. IBM proposes to use OpenMP to exploit parallelism in the Cell/B.E. processor [24]. OpenCL [17] allows to exploit tasks written in a C dialect for easy SIMDization, into the Cell/B.E., accelerators, and GPUs. Sequoia [11] exploits the memory hierarchy of accelerator-based machines. RapidMind [27] focusses in the exploitation of SIMD operations across regular data structures. Offload [9] targets easy spawning of code onto accelerators specially for gaming environments. StarSs [26] uses directives to express data needs for tasks, and derives the dependency tree at runtime to schedule the tasks appropriately across the accelerators. In the context of the SARC European project, we have proposed two different alternatives for programming accelerators: TPC* [34], and CellMP* [12]. In this paper, we mark these two proposals with the character '*' to reflect that they are the work done in the context of the SARC European project targeting support for heterogeneous architectures.

These efforts seem to converge in at least two common patterns for programming heterogeneous multi-core processors. The first, the work-offloading pattern, whereby one or more cores of the processor, preferably those cores that run the operating system and support a mature ISA (e.g. x86, PowerPC), off-load work to the rest of the cores, which typically have special acceleration capabilities, such as vector registers and execution units or fine-grain multithreading. The second pattern is the explicit management of locality, via control of data transfers from and to the accelerating cores, and via runtime support for scheduling these data transfers so that they can be overlapped with computation.

Despite convergence, very little is known about the relative performance of different programming models, the criticality of different programming constructs used in these models for high performance, and the implications of using these constructs for the programmer. To bridge this

	Concurrency constructs	Parallel execution constructs	Scheduling scheme	Locality control	Automatic function outlining
Sequoia	language types	tasks	static	explicit (language in/out data types)	no
StarSs	directives	tasks	dynamic	explicit (copy in/out directives)	no
CellGen	directives	parallel loops / tasks	static	implicit (compiler)	yes
TPC*	library calls	tasks	static	explicit (argument tags)	no
CellMP*	directives	parallel loops / tasks	static	explicit (copy in/out directives, blocking)	yes
Cell SDK	library calls	threads	static/dyn.	explicit (DMAs, buffers)	no

Table 1: Qualitative properties of the Cell/BE programming models evaluated in this study.

knowledge gap, we evaluate six programming models designed for and implemented on the Cell processor: Sequoia, StarSs, CellGen, TPC*, CellMP*, and the IBM provided low-level Cell SDK [14]. We make a first attempt at classifying programming models for heterogeneous multi-core processors in terms of the means for expressing parallelism and locality (directives vs. language types vs. runtime library calls), the vehicles of parallel execution (e.g. tasks, loops), their scheduling scheme (static, dynamic), their means for controlling and optimizing data transfers, and the availability of compiler support for automatic work-outlining. A succinct summary of this classification is shown in Table 1. We proceed to evaluate these models in a unified platform, using the same hardware (IBM Cell QS20 blades), same software stack (OS, back-end compilers) running beneath the programming models, and same applications. We evaluate both programming complexity and performance using a unified methodology across the different models.

— Draw a few conclusions. What is the complexity of programming? What is the cost of using high-level abstractions? What is the sensitivity of performance to the compiler and runtime environment? These conclusions should be based on the numbers and on comparing the quality of the generated code compared to the hand-tuned version of the benchmarks. —

2 Related Work

New computer architecture designs based on heterogeneous multicores have raised the question about their programmability. Bouzas et al [6] propose a MultiCore Framework (MCF) API toolkit that provides an abstract view of this hardware oriented toward computation of multidimensional data sets. The CAPS HMPP [10] toolkit is a set of compiler directives, tools and software runtime that supports multi-core processor parallel programming in C and Fortran on Unix platforms. HMPP works based on *codelets* that define functions that will be run in a hardware accelerator.

These codelets can either be hand-written for a specific architecture or be generated by some code generator.

There have been a number of publications on applications exploiting memory performance on the Cell BE processor. Among them, Saidani et al. [28] show the evaluation of several parallelization schemes for an image processing algorithm, comparing the results obtained from the Cell BE processor with other cache-based multicore architectures. Authors conclude that DMA based transfers offer more predictable results than cache-based data accesses. Benthin et al. [5] show how to efficiently map a ray tracing algorithm onto the Cell BE processor. They use software hyperthreading techniques to hide data access latencies, in a way comparable to double buffering. Their results show that a single SPU can offer performance comparable to an actual x86 CPU.

Recently, general purpose computation on graphic processors has received a lot of attention as it delivers high performance computing at rather low power. Major processor vendors have showed their intent to integrate GPUs as a GPU-core in the CPU chip [15, 1], so programming model techniques in this area are relevant in this discussion. CUDA [21], Compute Unified Device Architecture, proposed by the GPU vendor NVIDIA, is a programming model for General Purpose Graphic Processor Units (GPGPU) computing [22]. It is based on *kernels* that are run n times in parallel by n CUDA threads. These kernels are grouped in *blocks* and these in *grids*, creating a software hierarchy that should map perfectly to the memory hierarchy for better performance. For example, peak bandwidth performance of the NVIDIA GeForce 8800 GTX is 86.4 GBytes/s, around ten times the bandwidth that can offer a front side bus for an Intel CPU. But even this bandwidth performance is unable to sustain the 384 GFlops that the same card can achieve in floating point operations. For that reason mapping data to on-card local memories is of extreme importance. The CUDA programming environment is designed as an extension to C and C++ and also provides some pragmas in order to specify such mappings. Recently, tools to better map the algorithms to the memory hierarchy have been proposed [35]. They advocate for that programmers should provide straight-forward implementations of the application kernels using only global memory, and that tools like CUDA-lite will do the transformations automatically to exploit local memories. Brook for GPUs [32] is a compiler and runtime implementation of the Brook [31] stream program language that runs on programmable GPUs. Another alternative is MCUDA [33], that proposes to use CUDA as a data-parallel programming model on homogeneous multicore architectures. We think that our

proposal complements these ones, by allowing the programmer to express the parameters, like blocking factors, that usually are not so easy to find by compilers.

RapidMind [27] is a development and runtime platform that uses dynamic compilation to accelerate code for the accelerators available, being those GPUs or the Cell SPUs. The programmer encapsulates functions amenable for acceleration into program containers. The code in containers is only compiled during the execution of the application, so that it can be optimized dynamically depending on the input data and the target architecture.

Offload [9] is a programming model for offloading portions of C++ applications to run on accelerator cores. Code to be offloaded is wrapped in an *offload* block, indicating that the code should be compiled for an accelerator, and executed asynchronously as a separate thread. Call graphs rooted in an offload block are automatically identified and compiled for the accelerator. Data movement between host and accelerator memories is also handled automatically.

3 Applications

For our analysis, we use three applications: a memory bandwidth benchmark and two realistic supercomputer-class applications.

3.1 CellStream

We have used a memory bandwidth benchmark called CellStream to understand how to maximize SPE to main memory data transfers. It was designed so that a small computational kernel can be dropped in to perform work on data as it streams through SPEs. If no computational kernel is used, the benchmark is able to match the performance of the read/write DMA benchmark bundled with the Cell SDK 3.0.

For our experiments we use an input/output stream of 192Mbytes of data flowing across the SPUs. Data originates from a file, it is read in by a PPU thread, transferred in and out of one of the participating SPUs, and written by another PPU thread onto a separate file.

3.2 FixedGrid

Fixedgrid is a comprehensive prototypical atmospheric model written entirely in C. It describes chemical transport via a third order upwind-biased advection discretization and second order diffusion discretization [18, 29, 13]. An implicit Rosenbrock method is used to integrate a 79-species SAPRC’99 atmospheric chemical mechanism for VOCs and NO_x on every grid point [7]. Chemical or transport processes can be selectively disabled to observe their effect on monitored concentrations.

To calculate mass flux on a two-dimensional domain, a two-component wind vector, horizontal diffusion tensor, and concentrations for every species of interest must be calculated. To promote data contiguity, Fixedgrid stores the data according to function. The latitudinal wind field, longitudinal wind field, and horizontal diffusion tensor, are each stored in a separate $N_X \times N_Y$ array, where N_X and N_Y are the width and height of the domain, respectively. Concentration data is stored in a $N_S \times N_X \times N_Y$ array, where N_S is the number of monitored chemical species. To calculate ozone (O₃) concentrations on a 600×600 domain as in our experiments, approximately 1,080,000 double-precision values (8.24 MB) are calculated at each time step and 25,920,000 double precision values (24.7 MB) are used in the calculation.

3.3 PBPI

PBPI is a parallel implementation of the Bayesian phylogenetic inference method, which constructs phylogenetic trees from DNA or AA sequences using a Markov chain Monte Carlo (MCMC) sampling method. The computation time of a Bayesian phylogenetic inference based on MCMC is determined by two factors: the length of the Markov chains for approximating the posterior probability of the phylogenetic trees and the computation time needed for evaluating the likelihood values at each generation. The length of the Markov chains can be reduced by developing improved MCMC strategies to propose high quality candidate states and to make better acceptance/rejection decisions; the computation time per generations can be sped up by optimizing the likelihood evaluation and exploiting parallelism. PBPI implemented both techniques, and achieves linear speedup with the number of processors for large problem sizes.

For our experiments, we used a data set of 107 taxa with 19,989 nucleotides for a tree. There are

three computational loops that are called for a total of 324,071 times and account for the majority of the execution time of the program. The first loop accounts for 88% of the calls, and requires 1.2 MB to compute a result of 0.6 MB; the second loop accounts for 6% of the calls and requires 1.8 MB to compute a result of 0.6 MB; and the third also accounts for 6% of the calls and requires 0.6 MB to compute the weighted reduction of a vector, onto a result of 8 bytes.

4 Programming Models under Study

4.1 Handcoding on the Cell

Programming the Cell by hand requires use of the the Cell SDK 3.0, as provided by IBM. The SDK exposes Cell architectural details to the programmer, such as SIMD intrinsics for SPE code. It also provides libraries for low-level, Pthread style thread-based parallelization, and sets of DMA commands based on a get/put interface for managing locality and data transfers.

Programming in the Cell SDK is analogous, if not harder, than programming with MPI or POSIX threads on a typical cluster or multiprocessor respectively. The programmer needs both deep understanding of thread-level parallelization and deep understanding of the Cell hardware.

While programming models can transparently manage data transfers, the Cell SDK requires all data transfers to be explicitly identified and scheduled by the programmer. Furthermore, the programmer is solely responsible for data alignment, for setting up and sizing buffers to achieve computation/communication overlap, and for synchronizing threads running on different cores. However, hand-tuned parallelization also has well-known advantages. A programmer with insight into the parallel algorithm and the Cell architecture can maximize locality, eliminate unnecessary data transfers and schedule data and computation on cores in an optimal manner.

4.2 Sequoia

Sequoia expresses parallelism through explicit task and data subdivision. In this model, the programmer constructs trees of dependent tasks where the inner tasks call tasks further down the tree, eventually ending in a leaf task, which is typically where the real computation occurs. At each level, the data is decomposed and copied to the child tasks as specified. Each task has a private address space.

Locality is strictly enforced by Sequoia because tasks can only reference local data. In this manner, there can be a direct mapping of tasks to the Cell architecture where the SPE local storage is divorced from the typical memory hierarchy. By providing a programming model where tasks operate on local data, and providing abstractions to subdivide data and pass it on to subtasks, Sequoia is able to completely abstract away the underlying architecture from programmers. Sequoia allows programmers to explicitly define data and computation subdivision through a specialized notation. Using these definitions, the Sequoia compiler generates code which divide and transfer the data between tasks and performs the computations on the data as described by programmers for the specific architecture. The mappings of data to task and task to hardware are fixed at compile time.

Although programmers are theoretically free from the awareness of the underlying data transfer mechanism, the current Sequoia runtime system does not support transferring non-contiguous data. Sequoia tries to ensure that programmers are free from the awareness of the architectural constraints such as the DMA requirements of data alignment and size by providing an interface to allocate arrays. Programmers are expected to use such an interface to allocate arrays which are handled by Sequoia. For instance, the interface for Cell architecture decides the amount of data space to allocate as the sum of the multiple of 16 bytes larger than or equal to the requested size satisfying the DMA data size constraint and the additional padding for the DMA alignment constraint. Then, it allocates the space for the Sequoia array structure as well as the contiguous space for the array data pointed by the structure.

4.3 Star Superscalar (StarSs)

StarSs [25] is a task based parallel programming model. Similarly to OpenMP 3.0 the code is annotated with pragmas, although in this case the pragmas annotate when a function is a task. Another difference is that pragmas indicate the direction (input, output, or inout) of the parameters of the function, with the objective of giving hints to the runtime. This allows the StarSs runtime system to automatically discover the actual data dependencies between tasks. The StarSs runtime also implements data renaming, allowing to eliminate false dependencies in the task dependency graph.

4.4 CellGen

Cellgen implements a subset of OpenMP on the Cell [19]. The model uses a source-to-source optimizing compiler. Programmers identify parallel sections of their code in the form of loops accessing particular segments of memory. Programmers need to annotate these sections to mark them for parallel execution, and indicate how the data accessed in these sections should be handled. This model provides the abstraction of a shared-memory architecture and an indirect and implicit abstraction of data locality, via the annotation of the data set accessed by each parallel section. Note that while the data set accessed by each parallel section is annotated, the data set accessed by each task that executes a part of the work in the parallel section is not annotated.

Data is annotated as private or shared, using the same keywords as in OpenMP. Private variables follow OpenMP semantics. They are copied into local stores using DMAs and each SPE gets a private copy of the variable. Shared variables are further classified internally in the Cellgen compiler as *in*, *out*, or *inout* variables, using reference analysis. This classification departs from OpenMP semantics and serves as the main vehicle for managing locality on Cell. *In* data need to be streamed into the SPE's local store, *out* data needs to be streamed out of local stores, and *inout* data needs to be streamed both in and out of local stores. By annotating the data referenced in the parallel section, programmers implicitly tell Cellgen what data they want transferred to and from the local stores. The Cellgen compiler takes care of managing locality, by triggering and dynamically scheduling the associated data transfers.

Being able to stream *in/out/inout* data simultaneously in Cellgen is paramount for two reasons: the local stores are small, so they can only contain a fraction of the working sets of parallel sections; and the DMA time required to move data in and out of local stores may dominate performance. Overlapping DMAs with computation is necessary to achieve high performance. Data classified by the compiler as *in* or *out* are streamed using double-buffering, while *inout* data are streamed using triple buffering. The number of states a variable can be in determine the depth of buffering. *In* variables can be either streaming in, or computing; *out* variables can be either computing or streaming out; *inout* variables can be streaming in, computing, or streaming out. The Cellgen compiler creates a buffer for each of these states. For array references inside parallel sections, the goal is to maximize computation/DMA overlap by having different array elements in two (for *in*

and *out* arrays) or three (for *inout* arrays) states simultaneously.

SPEs operate on independent loop iterations in parallel. Cellgen, like OpenMP, assumes that it is the programmer's responsibility to ensure that loop iterations are in fact independent. However, scheduling of loop iterations to SPEs is done by the compiler. The current implementation uses static scheduling, where the iterations are divided equally among all SPEs.

4.5 Tagged Procedure Calls (TPC*)

TPC* [34] is a programming model that exploits parallelism through asynchronous execution of tasks. The TPC* runtime system creates a task using a function descriptor and an arguments descriptor as input and identifies the task using a unique task ID. The programmer uses library calls to identify certain procedure calls as concurrent tasks and specify properties of the data accessed by them, to facilitate their transfers to and from local memories. Each argument descriptor is formed in a triplet containing the base address, the argument size, and the argument type that can be either IN, OUT or INOUT. The TPC* runtime handles contiguous arguments and strided arguments with a fixed stride. For the latter, the programmer masks the type field with the STRIDE flag and packs the number of strides, stride size and offset within the size field with the help of a macro. The sizes of task arguments define the granularity of parallelism within a region of straight-line code or a subset of the iteration space of a loop. The programmer implements synchronization by using either point-to-point or collective wait primitives.

The PPE issues tasks across SPEs round robin, using remote stores. Each active SPE thread has its own private queue that the issuer can access to post tasks. Each SPE runs a thread that continuously polls its local queue and executes any available tasks in FIFO order. For each new task, the main thread running on PPE tries to find an empty slot in some SPE queue to issue the task. Upon task completion each SPE thread updates the task status in a thread-specific completion queue located in the PPE cache. In this case, to avoid cache invalidation and the resulting off-chip transfer, the TPC* runtime uses atomic DMA commands to unconditionally update the PPE's cache. PPE polls these completion queues to detect task completion. This polling is done when no more tasks can be issued or a synchronization primitive have been reached.

The TPC* runtime uses only on-chip operations when initiating and completing tasks, whereas argument data may require off-chip transfers. The runtime uses task argument pre-fetching and

outstanding write-backs to overlap communication with DMA latency.

4.6 CellMP*

CellMP* uses the OpenMP 3.0 *task* approach to express the work to be executed in the accelerators. To distinguish among OpenMP code for the regular processors, and code targeting an accelerator, we have proposed a new clause `device(accelerator-kind-list)` [12].

Also in our proposal, the use of the `copy_in(variable | array-section)` clause causes a transfer of data from main memory to the accelerator; `copy_out(variable | array-section)` will similarly cause the variable or array-section to be written back from the accelerator to main memory. Since local memories have limited size, we might want to split the parallel computation to work on chunks of the main memory arrays, in particular for the situations where the whole array or matrix does not fit in the accelerator memory. In this case, the specification of `array-sections`, in a way inherited from Fortran90 is quite convenient. Communication is implemented synchronously, and to support communication-computation overlap we use CellMT threads [4, 3]. Usually, two SPU threads are created in each SPU, so that while one of them is computing the other one is in the data transfer stage.

Loop blocking [20, 16, 36] is also used as a way to coarsen work and to overcome memory constraints of the auxiliar processors. The scope of blocking includes the N loops surrounding the enclosed loop body and it is defined by a clause `factors (F_1, F_2... F_N)` to specify the blocking factor of each loop in the considered loop nest, starting from the outermost one. The reader can refer to [12] for code examples showing the use of CellMP* directives.

5 Evaluation

In this section, we evaluate first the memory bandwidth achieved by each of the programming models using CellStream, and then we present the evaluation of the two realistic applications, Fixedgrid and PBPI.

file	type	change	SDK	Sequoia	CellGen	StarSs	TPC*	CellIMP*
main.c	Offloading fn	+ s	29	7	7	16	10	19
		+ p	-	-	1	2	-	8
		- s	2	2	3	5	18	7
spe_main.c	SPE task/kernel	+ s	81	46	-	-	27	-
		- s	0	0	-	-	0	-
Total	All lines	+	110	53	8	18	37	27
		-	2	2	3	5	18	7
	User functions added	+	1 task	1 task + 1 leaf	none	none	1 task	none

+ s: sentences added + p: pragmas added - s: sentences removed

Table 2: Changes in line counts in CellStream

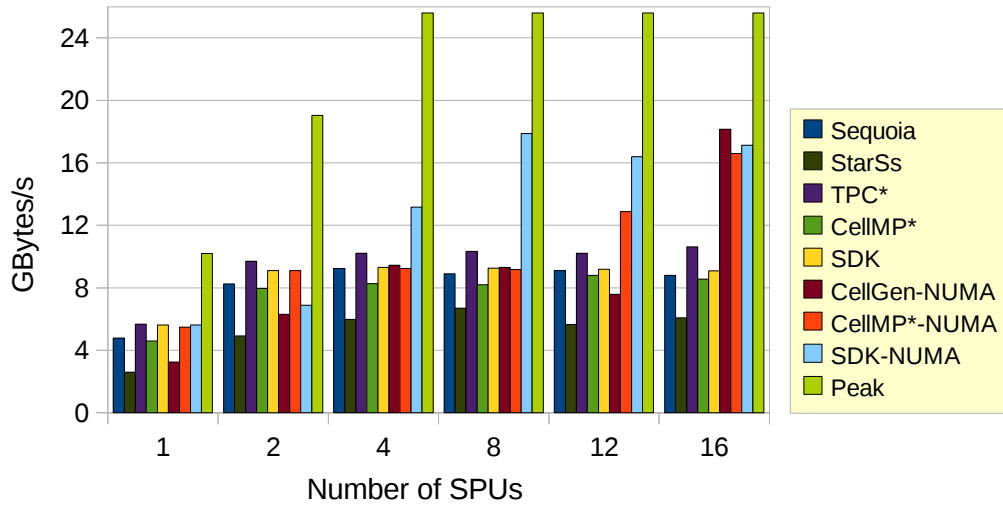


Figure 1: Memory bandwidth (GBytes per second) obtained from CellStream

5.1 CellStream

Table 2 shows, for each one of the programming models under evaluation, the number of sentences added/removed and pragmas added into CellStream in order to be able to run it using the Cell SPUs as accelerators. All changes have been done manually, before compiling the application with the corresponding programming model.

Porting the benchmark to run on the SDK is the task that introduces more changes into it (need to add more than 100 lines). Sequoia needs roughly half of the additions, and TPC* reduces them to one third. CellGen, StarSs and CellMP* require one fourth or less changes, due to the use of pragmas. For Sequoia, the programmer needs to write two functions for each parallel region, one (non-leaf) containing a sentence (`mappar`) to map the leaf function into an SPU. In the case of StarSs, and CellMP*, the changes are very similar, mainly because the code to outline is already encapsulated in a function in the serial version. In CellMP*, we account for each task/copy_in/copy_out/block line separately. This and the fact that CellMP* allows to express the block transformation with the pragmas, are the reasons why CellMP* has usually a higher number of pragmas added than the other models. CellGen is closer to OpenMP, and this is the reason why the changes in the original code are lower in number.

Figure 1 shows the performance obtained in CellStream in GBytes/s bandwidth. As the benchmark is limited by the memory bandwidth, and the Cell blades have two distributed memory banks, we present the evaluation of the programming models with no special memory allocation, and with NUMA memory allocation. The plot shows neatly the benefits of using NUMA capabilities from the runtime systems of CellGen, CellMP, and the SDK. For StarSs, we have used interleaved memory, with every other page in the same node. In this environment, we can not control the data-to-SPU mapping due to its dynamic nature, and the performance achieved is limited. The four models with static scheduling achieve roughly the same performance. Differences are attributed to the different implementation of the runtime systems. CellGen-NUMA, CellMP*-NUMA and SDK-NUMA behave differently depending on the number of SPUs used, because of the different NUMA management. For example, in SDK-NUMA, the SPUs are always evenly distributed across the two Cell chips. Instead, in CellMP*-NUMA, the runtime system uses what the Linux kernel offers at each execution, and for the experiments on 8 SPUs and less, all SPUs belong to the same Cell chip.

file	type	change	SDK	Sequoia -scalar	Sequoia -SIMD	GellGen	TPC* SIMD	CellMP* -SIMD
transport.c (103 lines)	Offloading fn	+ s	42	27	27	32	95	40
		+ p	-	-	-	2	-	26
		- s	33	15	33	11	14	11
transport_spe.c	SPE task	+ s	349	0	0	-	-	-
discretize.c (143 lines)	SPE kernel	+ s	219	119	314	0	51	0
		- s	60	0	60	0	0	1
Total	All lines	+	610	146	341	34	146	66
		-	93	15	93	11	14	12
	User functions added	+	2 tasks	2 tasks + 2 leaf	2 tasks + 2 leaf	none	2 tasks	none
+ s: sentences added + p: pragmas added - s: sentences removed								

Table 3: Changes in line counts in Fixedgrid

This forbids to achieve the full memory bandwidth available. The fact that CellGen-NUMA in 16 SPUs beats the SDK-NUMA implementation is also attributed to the different characteristics of the runtime system.

5.2 Fixedgrid

Table 3 shows the changes done in Fixedgrid. Again, SDK and Sequoia require the higher number of changes, with SDK growing up to 610 lines added. In both, this is also due to an extra transformation of the code done to improve DMA transfers. TPC* with the manual changes for SIMDization gets equivalent to the scalar version of Sequoia, meaning that it is simpler to write applications with TPC*, than with Sequoia. CellGen, and CellMP take advantage of the pragmas to further reduce the amount of changes required. Again, with CellMP* the number of annotations needed is higher than in CellGen. In this case, this is due to the pragmas used to offload 3 matrix transpositions to the SPUs, which is not done with CellGen.

Figure 2 shows the performance obtained from Fixedgrid. Serial (running on the PPU) and SDK-SIMD versions are shown as a reference. The SDK-SIMD and Sequoia-SIMD versions fully vectorize the row discretization performed in Fixedgrid. With this, in these models, the data transfers and the SIMD operations are completely optimized. Then, all the programming models scale nearly the same, except for CellGen, due to the lack of offloading of the data transpositions mentioned above. For TPC* and CellMP*, we have tried two different kernel codes. The version *gccSIMD* has been vectorized automatically by the gcc compiler; the *OPTK* version has been hand-tuned, and branches mostly eliminated in the inner-most function of the kernel. The results show that optimizing different aspects of the code can be as important as doing a fully SIMDization.

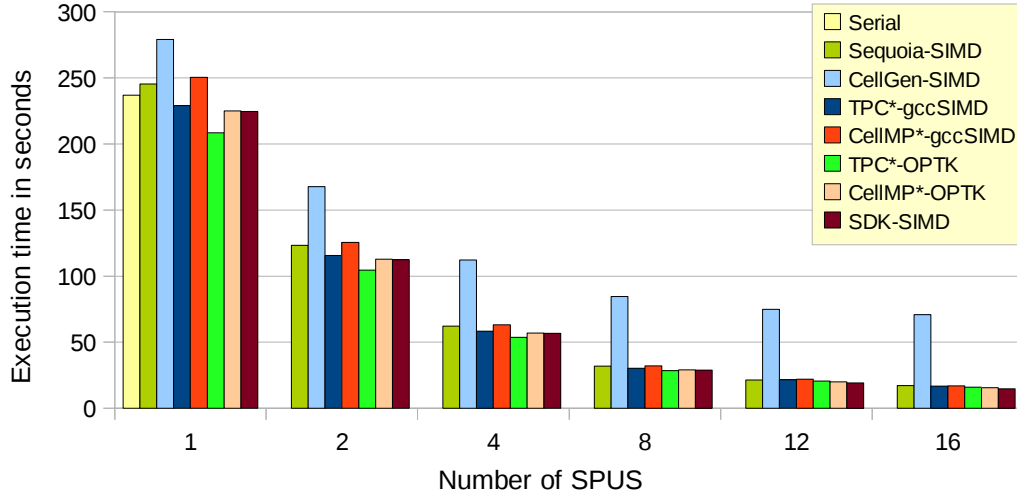


Figure 2: Execution times obtained in Fixedgrid

5.3 PBPI

Table 4 summarizes the changes in source code for PBPI. For CellGen-SIMD, StarSs-SIMD and CellIMP*-SIMD, we provide separate line counts for each of the kernels (rows labeled L[123]_spe.c), although they are not added as separate functions, but they just replace the scalar code used in the scalar versions. In this case, SDK, Sequoia and TPC*-SIMD are the models that need more changes to the original code. For TPC*-SIMD, the reason for the 215 lines added is that each task has a version running in both the PPU and the SPUs. Basically, those lines are the code to spawn TPC* on the PPU and the SPUs. The rows labeled L[123]_spe.c contain the sum of the lines needed for the tasks in the PPU and the SPU. As usual, the models based on pragmas require the use of less changes. It is interesting to note, that if the native compiler could vectorize the kernels automatically, this would have saved in the order of 150's lines.

Figure 3 shows the performance obtained from each of the programming models. It shows the serial, scalar and SIMD, and the SDK versions. In PBPI, tasks are small enough to stress the PPU spawning mechanism. In this sense, CellGen, CellIMP* and the SDK versions exploit the parallelism in a way similar to OpenMP parallel for loops. This is the reason for the higher performance obtained. Exploiting parallel loops in this way is important to create less tasks. Having to create less tasks puts less pressure on the PPU processor. This requires that each task can bring data to work with during runtime. This is achieved in CellIMP* with the possibility of expressing

File	type	Change	SDK-scalar	SDK-SIMD	Sequoia-scalar	Sequoia-SIMD	CellGen-SIMD	StarSs-scalar	StarSs-SIMD	TPC*-SIMD	CellIMP*-scalar	CellIMP*-SIMD
likelihood.c (463 lines)	Offloading funcs	+ s	32	32	19	19	14	123	64	215	17	17
		+ p	-	-	-	-	3	4	4	-	19	19
		- s	52	52	54	54	70	58	58	13	2	115
pbpi_spe.c	task/kernel	+ s	200	219	223	170	-	-	-	48	-	-
L1_spe.c	kernel	+ s	18	28	0	56	28	0	60	71	0	55
L2_spe.c	kernel	+ s	25	41	0	72	41	0	74	94	0	74
L3_spe.c	kernel	+ s	17	21	0	53	21	0	24	26	15	60
Total	All lines	+	292	341	242	370	104	123	222	454	32	206
		-	52	52	54	54	70	58	58	13	2	115
	User functions added	+	3 tasks	3 tasks	3 tasks + 3 leaf	3 tasks + 3 leaf	3 tasks	3 tasks	3 tasks	3 PPU/3 SPU	none	none

+ s: sentences added + p: pragmas added - s: sentences removed

Table 4: Changes in line counts in PBPI

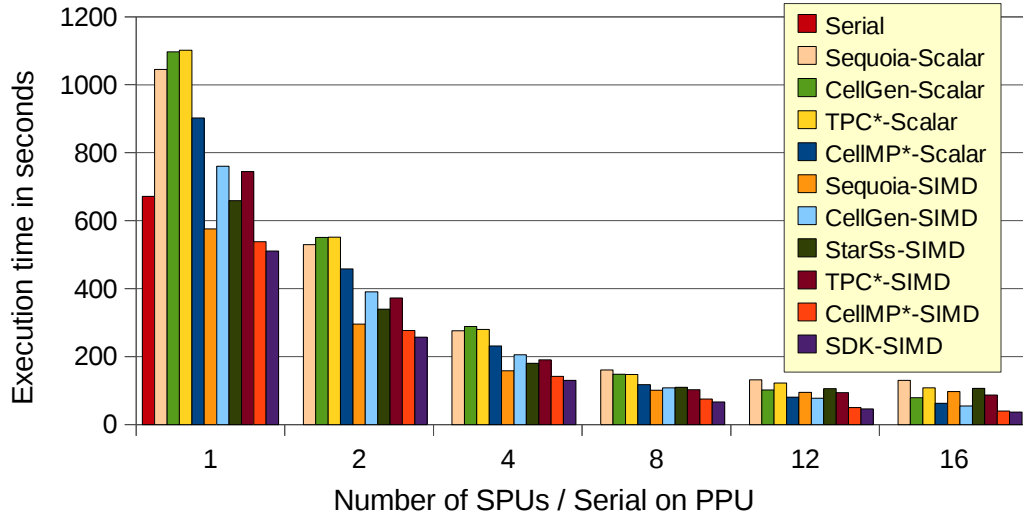


Figure 3: Execution times obtained in PBPI

copy_in/copy_out data movements in the middle of the tasks code, not only at the beginning and the end. Even with this approach, CellIMP*-SIMD is still 4 seconds slower than the SDK version when using 16 SPUs.

6 Conclusions and Future Work

Acknowledgements

This work has been supported by the European Commission in the context of the SARC European IP project (contract no. 27648), the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the Spanish Ministry of Education (contract no. TIN2007-60625), by the Generalitat de Catalunya (2009-SGR-980), and the BSC-IBM MareIncognito project.

References

- [1] AMD Corporation. AMD 2007 Technology Analyst Day. http://www2.amd.com/us-en/assets/content_type/DownloadableAssets/FinancialA-DayNewsSummary121307FINAL.pdf.
- [2] Kevin J. Barker, Kei Davis, Adolfo Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jos Carlos Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC*, page 1. IEEE/ACM, November 2008.
- [3] Vicenç Beltran, David Carrera, Jordi Torres, and Eduard Ayguadé. Cooperative Multithreading on the Cell BE. Technical report, Computer Architecture Department, Technical University of Catalonia, April 2009.
- [4] Vicenç Beltran, David Carrera, Jordi Torres, and Eduard Ayguadé. CellMT: A Cooperative Multithreading Library for the Cell /B.E. In *HiPC 2009: Proceedings of the 16th Annual IEEE International Conference on High Performance Computing*. IEEE Computer Society, December 2009.
- [5] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the cell processor. In *IEEE Symposium on Interactive Ray Tracing*, 2006.
- [6] B. Bouzas, R. Cooper, J. Greene, M. Pepe, and M. J. Prella. MultiCore Framework: An API for Programming Heterogeneous Multicore Processors. In *Proceedings of First Workshop on Software Tools for Multi-Core Systems*, New York, NY, USA, 2006. Mercury Computer Systems.
- [7] William P. L. Carter. Documentation of the saprc-99 chemical mechanism for voc reactivity assessment. Final Report Contract No. 92-329, California Air Resources Board, May 8 2000.
- [8] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell Broadband Engine Architecture and its first implementation. *IBM DeveloperWorks*, November 2005.
- [9] Pete Cooper, Uwe Dolinsky, Alastair F. Donaldson, Andrew Richards, Colin Riley, and George Russell. Offload – automating code migration to heterogeneous multicore systems. In *Lecture Notes in Computer Science, HiPEAC Conference 2010*, pages 307–321, 2010.

- [10] Romain Dolbeau, Stphane Bihan, and Franois Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on General Processing Using GPUs*, 2006.
- [11] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *SC '06: Proceedings of the 2006 Conference on High Performance Networking and Computing (Supercomputing'2006)*, pages 83–92, 2006.
- [12] Roger Ferrer, c Beltran Vicen Marc Gonzalez, Xavier Martorell, and Eduard Ayguade. Analysis of task offloading for accelerators. In *Lecture Notes in Computer Science, HiPEAC Conference 2010*, pages 322–336, 2010.
- [13] W. Hundsdorfer. Numerical solution of advection-diffusion-reaction equations. Technical report, Centrum voor Wiskunde en Informatica, 1996.
- [14] IBM. Cell Broadband Engine resource center, 2008.
<http://www.ibm.com/developerworks/power/cell/downloads.html>.
- [15] Intel Corporation. Intel Corporation’s Multicore Architecture Briefing, March 2008.
<http://www.intel.com/pressroom/archive/releases/20080317fact.htm>.
- [16] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [17] Khronos OpenCL Working Group. The OpenCL Specification, February 2009.
<http://www.khronos.org/registry/cl/>.
- [18] John C. Linford and Adrian Sandu. Optimizing large scale chemical transport models for multicore platforms. In *Proceedings of the 2008 Spring Simulation Multiconference*, Ottawa, Canada, April 14–18 2008.
- [19] Tim Mattson. Introduction to openmp. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 209, New York, NY, USA, November 2006. ACM.

- [20] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.
- [21] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Version 2.0*, 2008.
- [22] NVIDIA Corporation. *NVIDIA Tesla GPU Computing Technical Brief*, 2008.
- [23] NVIDIA Corporation. *NVIDIA Fermi: Next Generation CUDA Architecture*, 2009.
- [24] Kevin O’Brien, Kathryn O’Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting OpenMP on Cell. In *International Journal of Parallel Programming*, 2008.
- [25] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. *IEEE Int. Conference on Cluster Computing*, pages 142–151, September 2008.
- [26] Josep M. Perez, Pieter Bellens, Rosa M. Badia, and Jesus Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5):593–604, September 2007.
- [27] RapidMind. RapidMind Multi-core Development Platform. <http://www.rapidmind.com/pdfs/RapidmindDatasheet.pdf>.
- [28] Tarik Saidani, Stéphane Piskorski, Lionel Lacassagne, and Samir Bouaziz. Parallelization schemes for memory optimization on the cell processor: a case study of image processing algorithm. In *MEDEA ’07: Proceedings of the 2007 workshop on MEMory performance*, pages 9–16, New York, NY, USA, 2007. ACM.
- [29] A. Sandu, D.N. Daescu, G.R. Carmichael, and T. Chai. Adjoint sensitivity analysis of regional air quality models. *Journal of Computational Physics*, 204:222–252, 2005.
- [30] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

- [31] Stanford University. Brook Language.
<http://merrimac.stanford.edu/brook/>.
- [32] Stanford University. BrookGPU.
<http://graphics.stanford.edu/projects/brookgpu/>.
- [33] John A. Stratton, Sam S. Stone, and Wen mei W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs. In *In Languages and Compilers for Parallel Computing (LCPC) 21st Annual Workshop*, August 2008.
- [34] George Tzenakis, Konstantinos Kapelonis, Michail Alvanos, Konstantinos Koukos, Dimitrios S. Nikolopoulos, and Angelos Bilas. Tagged procedure calls (tpc): Efficient runtime support for task-based parallelism on the cell processor. In *Lecture Notes in Computer Science, HiPEAC Conference 2010*, pages 307–321, 2010.
- [35] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen mei W. Hwu. CUDA-lite: Reducing GPU Programming Complexity. In *In Languages and Compilers for Parallel Computing (LCPC) 21st Annual Workshop*, August 2008.
- [36] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.