Application Porting and Tuning on the Cell-BE Processor

Martin Rehr and Brian Vinter

Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
{rehr, vinter}@diku.dk

Abstract. The Cell is a heterogeneous multi-core processor, consisting of 9 cores with a peak performance in excess of 100 giga-operations per second. To make the Cell processor provide more that a fraction of that performance, a very high level of parallelism is needed as well as a number of basic, but very important code optimizations. This paper covers the process of porting an existing recursive application to the Cell processor, and the steps needed to achieve high performance. Optimization methods such as the four levels of parallelism supported by the Cell, task-, memory-, data-, and instruction-level parallelization are covered as well as branch elimination. The final performance numbers show that the Cell processor outperforms traditional processor architectures quite impressingly if an application is ported properly.

1 Introduction

Intel cancelled the Pentium 4, 4 GHz in 2004 due to heating issues, we still haven't seen it and never will. Moore's law[6] is still going strong, which means the number of transistors per square-inch doubles approximately every two years. Using the increasing amount of transistors for increasing cache-sizes and deeper pipelines are no longer improving performance significantly. This has led to the design of multi-core processors, some of which are just extensions of the old processor design making it easy to apply existing applications to them, but making it hard to gain any significant performance increase. However Sony, IBM and Toshiba have developed the Cell Broadband Engine (Cell BE[4]) which is a powerful heterogeneous multi-core chip capable of doing 204 GFLOPS single precision and 14 GFLOPS double precision, using a whole new architecture. To get near this kind of performance one has to carefully tune the applications towards this architecture.

1.1 The Cell BE

The Cell processor is a heterogeneous multi core processor consisting of 9 cores, The Primary core is an IBM 64 bit power processor (PPC64) with 2 hardware threads. This core is the link between the operating system and the 8 powerful

working cores, called the SPE's for Synergistic Processing Element. The power processor is called the PPE for Power Processing Element, all cores are connected by an Element Interconnect Bus (EIB). Figure 1 shows an overview of the Cell architecture. The cores are connected by an Element Interconnect Bus (EIB)

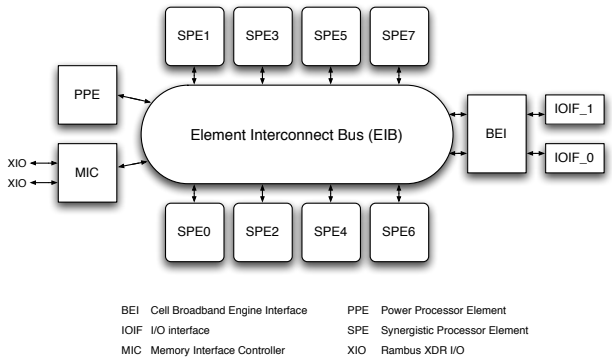

Fig. 1. An overview of the Cell architecture

capable of transferring up to 204 GB/s at 3.2 GHz. Each SPE is dual pipelined, has a 128x128 bit register file and 256 kB of on-chip memory called the local store. Data is transferred asynchronously between main memory and the local store through DMA calls handled by a dedicated Memory Flow Controller (MFC). An overview of the SPE is shown in figure 2. By using the PPE as

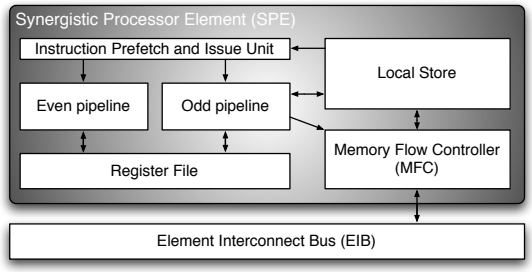

Fig. 2. An overview of the SPE cores

primary core, the Cell processor can be used out of the box, due to the fact that existing operating systems support the PPC64 architecture. Thereby it's possible to boot a PPC64 operating system on the Cell processor, and execute PPC64 applications, however these will only use the PPE core. To use the SPE

cores it's necessary to develop code specifically for the SPE's, which includes setting up a memory communications scheme using DMA through the MFC.

2 Porting towards the Cell

To make the Cell BE perform at a high level one has to consider several levels of parallelization, including task-, memory-, data-and instruction-level parallelization. As an experiment an nqueen[2][5] application written by Takaken[9] has been ported from the X86 architecture to the Cell architecture. The nqueen problem is solved by using a divide and conquer algorithm for finding how many ways to place N queens safely at an NxN chess board according to the common chess rules.

2.1 Task-and memory-parallelization

As the Cell BE architecture can be viewed as an 8 node (the SPE's) cluster[7] on a chip with a front end (The PPE) splitting an application into smaller tasks is done by the same principles as when parallelizing towards a cluster computer. However due to the limited amount of local store (256 kB) available at the SPE's for both code and data, one has to consider the size of each task. This means that an application which would be best parallelized by a bag of task model in a cluster setup, might be best parallelized by a pipelined setup using the Cell processor.

2.1.1 Task-parallelization The first step of porting an application to the Cell is to parallelize it task wise, using the PPE as a task manager and the SPE's as computation units. This is done by analyzing the application, picking the right method for parallelization including analyzing if data and code for each task fits into the 256 kB local store of the SPE's, and if not, which method one wishes to use to make it fit. The two possibilities here are either to split the data-set for each task into smaller data-sets or to use a pipelined setup, where the code is split among 2,4 or 8 SPE's each performing some piece of the computation.

2.1.2 Memory-parallelization Each SPE has its own MFC controller running in its own thread, meaning that main memory can be accessed through DMA asynchronously regarding computation. This gives the possibility of effective memory latency hiding, as the MFC writes data directly from the EIB to the local store without involving the computational unit. Thereby the data for iteration i+1 can be retrieved while computing iteration i, this is known as double buffering.

Furthermore each MFC is capable of issuing 16 simultaneous DMA transfers giving a high level of possible multi-buffering[1]. When porting an application,

[1] Where data for iteration i+1,i+2, ..., i+n is retrieved in iteration i

a communication scheme between the PPE and the SPE using DMA transfers through the MFC has to be chosen. Most applications will use at least double buffering to hide memory latency, and some applications need to use multi-buffering to keep the computational unit busy, this all depends of the computational intensiveness of the task.

2.1.3 The nqueens solution In the nqueens example, code and data fit into the local store of each SPE, and therefore a bag-of-task model is used, where each SPE requests a task from the PPE, gets the input data, computes the result, delivers the result to the PPE and requests a new task. As the input and output data for the nqueens application is quite small and the compute intensive part of the application is quite large, double buffering is sufficient for keeping the SPE's busy, hiding the memory latency efficiently. This first step reduced the execution time of placing 18 queens on an 18x18 chess board from 278.878 seconds on a Pentium 4 running at 3,2 GHz to 69.456 seconds when executed on a Playstation 3[2] giving a speedup of 4. The application has been compiled with both the GCC compiler and IBM's XLC compiler, the result is shown in figure 3. It's seen that in this case the GCC compiler produces code which is

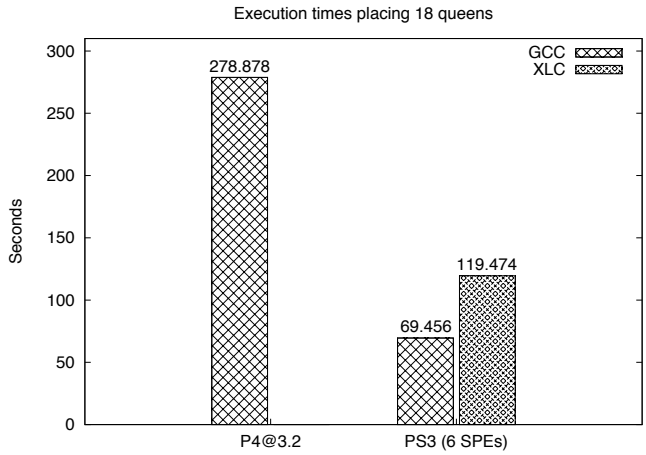

Fig. 3. Execution times for the task parallelized nqueens application

significantly faster than the code produced by the XLC compiler.

[2] Note the PS3 only has 6 SPE's

2.2 Register-line optimizations

The Cell BE SPE's are SIMD[8] vector cores each operating on a 128 bit register-line, which can be divided into 2x64 bit longs, 4x32 bit int's, 8x16 shorts or 16x16 bytes. This results in scalar operations to be mapped down to an atomic sequence of register-line operations, as the scalar has to be put in its preferred slot of the register-line see figure 4. The compute intensive part of the code should be fitted

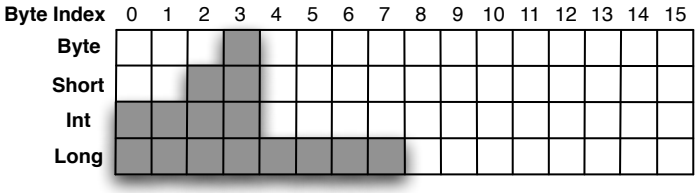

Fig. 4. The preferred slot in a register line for the different data types

to use 128 register-line operations instead of scalars, as this will eliminate the rotate and shuffle instructions needed to get the scalar into the correct preferred slot.

2.2.1 Recursive vs. iterative methods If the application to be ported is of a recursive nature, one might consider to transform the compute intensive part into an iterative method, as the limited local store of 256 kB is exhausted quite quickly if deep recursions are reached. Furthermore, the use of recursive methods might slow down execution if the data used in the compute intensive part is fitted into the 2 kB register file, as local parameters are pushed to the stack upon a function call.

2.2.2 Branch prediction and elimination The Cell processor has no hardware branch predictor, but the instruction set contains a branch hinting instruction. However if the programmer has no clue on what branch is to be taken, and the piece of code within each branch is fairly small, branch elimination can be done by calculating both results and selecting the right result based on the branch condition[3]. This has two advantages, it eliminates branch misses and it operate directly on register-lines whereas normal branch operations operates on scalars. Using these two methods the overall penalty of branching can be reduced quite impressively.

2.2.3 The nqueens solution The compute intensive part of the presented nqueen application was transformed from a recursive algorithm, to an iterative algorithm using the described register-line optimizations and branch elimination techniques. This resulted in an execution time reduction from 69.456 seconds

to 42.486 seconds, giving a speedup of 1.63 compared to the task parallelized code, and a speedup of 6.55 compared with the Pentium 4 execution running at 3.2 GHz. Furthermore the performance of the iterative scalar version of the algorithm was measured, all versions was compiled with both the GCC and XLC, the results are shown in figure 5. It's seen that the recursive scalar code

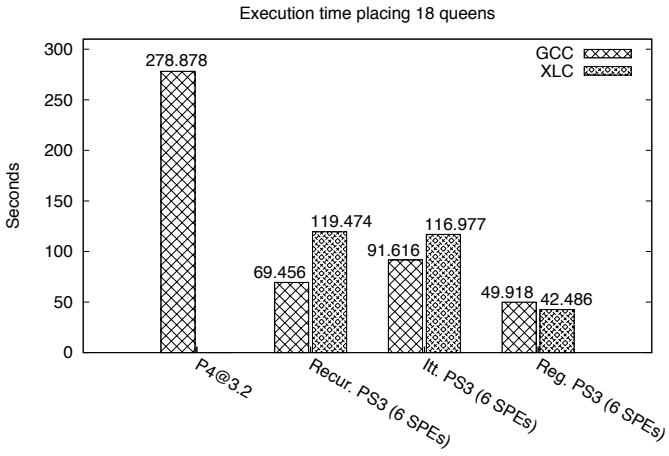

Fig. 5. Execution time comparisons between the different optimizations

overall performs a little better than the scaler iterative code, but that the XLC compiler produces faster binaries when used on the iterative code. Finally it's seen that the XLC compiler produces 15% faster binaries from the register-line code compared to GCC. These results show that the GCC produces faster binaries in the common case[3] whereas XLC produces faster binaries when the application is tuned towards the Cell SPE's architecture.

2.3 Data parallelization

As mentioned, the Cell BE SPE cores are SIMD vector cores, which offer data parallelization as an optimization parameter. Each SPE core is capable of performing 4 integer operations, 8 short operations or 16 byte operations per instruction. If the data has an integer SIMD nature, for example integer matrix multiplication, one can effectively vectorize by loop-unrolling, doing four integer operations in each loop shortening the total loop length by a factor of four. Otherwise if the application has a divide and conquer nature, one can vectorize by doing four branches simultaneously. The performance gain of branch vectorization depends heavily on the balance of the four branches, as the amount of leaves traversed is the worst case of the four branches.

[3] Applications written for traditional single core architectures

2.3.1 The nqueens solution The compute intensive part of the presented nqueens application is based on a divide and conquer algorithm. This algorithm was vectorized by investigating four branches simultaneously. The execution time of the vectorized code placing 18 queens was reduced from 42.486 seconds to 41.248 seconds, which is considered an insignificant speedup, the result is shown in figure 6. An analysis of the vectorized code showed that the average depth

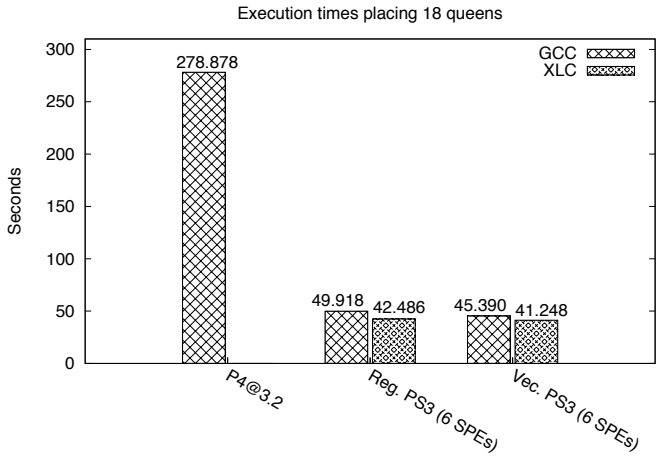

Fig. 6. Execution times for the vectorized nqueens application

reached in the search tree increased by 0.3 due to unbalanced branch vectorization. The presented nqueens application has a search space of N!, this means that solving the problem for more than 14 queens eliminates the advantage of branch vectorization. Performance measurements show that when placing 20 queens, the branch vectorized code gets slower than the register-line optimized code, figure 7. This is due to additional operations needed to test when all four branches within the vector have met their termination criteria. To make the branch vectorized code perform better, one could try to balance the branches placed within each vector. This has not been done with the presented nqueens application, as it's believed by the authors that the amount of processing power needed to balance the branches within the vectors is equal, or exceeds, the processing power needed to solve the problem itself.

2.4 Instruction parallelization

As the Cell BE is dual pipelined, one pipeline for computation and one for management, it's possible to perform load/store instructions from/to the local store while a computation instructions is being performed, this is specially usable within loops over data-sets, where it's clear which data are needed next. This is one of the optimization tasks which the compiler is capable of doing.

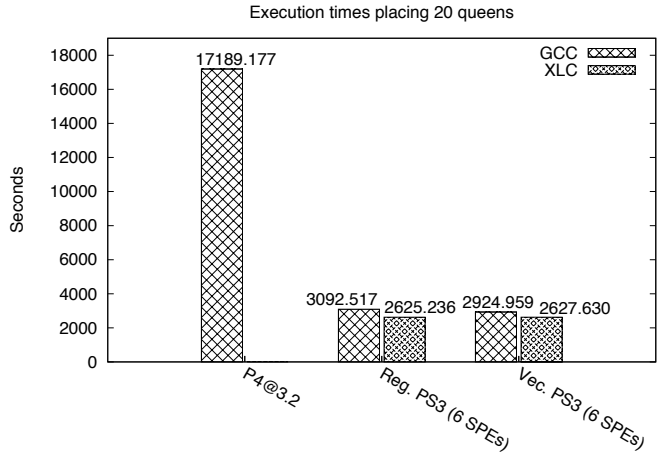

Fig. 7. Execution times for the register-line and branch-vectorized nqueens

2.4.1 The nqueens solution To measure how effectively the compiler interleaves the computation and data management instructions, the iterative register-line version[4] of the nqueens application compiled with the IBM XLC compiler has been profiled by using the Cell-BE system simulator[1]. The result of this was that approximately 13.9% of all instructions are instruction parallelized[5]. However the profiling revealed that 8.5% of all clock-cycles are used waiting on load/stores between the register-file and the local store. This indicates there is still room for improvement, which can be done either by eliminating local store access by using registers if possible, or doing instruction level parallelizing by hand at assembler level. This has not been looked further into.

2.5 Summary

When porting an application from a traditional single core application, i.e. the X86, to the Cell BE architecture, at least task-and memory-parallelization should be chosen as the PPE of the Cell in itself performs poorly, see figure 8. The core computational part of the task-and memory-level parallelized application can be compiled directly on the SPE's and thereby requires no rewriting. As mentioned, the speedup gained using the task and memory-level parallelization is 4 on 6 SPE's. Moving from the tasks-and memory-level parallelized version to the register-line version with branch elimination improved the speedup from 4 to 6.5 using 6 SPE's, but this required a rewrite of the compute intensive part using 203 lines of code opposed to the 75 lines of code used in the original recursive version. The vectorized version resulted in 348 lines of code, but didn't perform at all, due to unbalanced branch-vectors, however if one has a well balanced

[4] The iterative register-line version was the one performing best
[5] Called dual-instructions in Cell-BE terminology

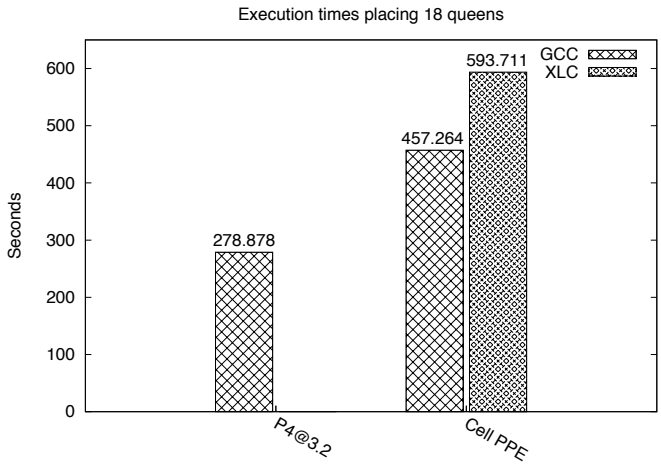

Fig. 8. Execution times for the PPE nqueens application

branch-vector or data is SIMD applicable one can expect a vectorized version to perform well, but it's not possible to reach beyond a speedup of 128/X, where X is the size of each data entry in the vector, compared to a register-line version.

3 Conclusion

The basic principles of porting an X86 application towards the Cell BE architecture has been presented in this paper as well as tuning methods including memory-, data-, and instruction-level parallelization. The porting of the nqueens test application shows that the execution time for placing 18 queens on an 18x18 chess board was reduced from 278.878 seconds on a Pentium 4 running at 3.2 GHz to 69.457 seconds, yielding a speedup of 4 just by applying standard task- and memory-parallelization techniques known from cluster computing. This was achieved without rewriting the compute intensive part towards the Cell BE architecture. Rewriting the compute intensive part towards the Cell-BE architecture reduced the execution time of placing 18 queens to 42.486 seconds, yielding a speedup of 6.5 compared to a Pentium 4 running at 3.2 GHz. The vectorized version of the nqueens application didn't yield any speedup as the average depth reached in the search tree increased by a factor of 0.3 neglecting the performance gain reached by investigating the four branches simultaneously. To gain speedup by vectorizing, the vector-branches need to be balanced, or the data needs to have a SIMD nature.

References

1. IBM Full-System Simulator for the Cell Broadband Engine Processor. http://www.alphaworks.ibm.com/tech/cellsystemsim.
2. *Berliner Schachgesellschaft*, 3:363, 1848.
3. Cell broadband engine programming handbook. pages 891–703, 2007. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/9F820A5FFA3ECE8C8725716A0062585F/$file/CBE_Handbook_v1.1_24APR2007_pub.pdf.
4. Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation. *IBM developerWorks*, 2005. http://www.ibm.com/developerworks/power/library/pa-cellperf.
5. Edsger W. Dijkstra. Chapter i: Notes on structured programming. pages 72–82, 1972.
6. Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38,8, 1965. ftp://download.intel.com/research/silicon/moorespaper.pdf.
7. T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.
8. Michael Sung. SIMD parallel processing. 2000. http://www.ai.mit.edu/projects/aries/papers/writeups/darkman-writeup.pdf.
9. Takaken. http://www.ic-net.or.jp/home/takaken/e/queen/index.html.