Can We Apply Accelerator-Cores to Control-Intensive Programs?

Sean Rul Hans Vandierendonck Koen De Bosschere

Department of Electronics and Information Systems (ELIS,
Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
E-mail: {srul, hvdieren, kdbosche}@elis.ugent.be

Abstract

There is a trend towards using accelerators to increase performance and energy efficiency of general-purpose processors. So far, most accelerators have been build with HPC-applications in mind. A question that arises is how well can other applications benefit from these accelerators?

In this paper, we discuss the acceleration of three benchmarks using the SPUs of a Cell-BE. We analyze the potential speedup given the inherent parallelism in the applications. While the potential speedup is significant in all benchmarks, the obtained speedup lags behind due to a mismatch between micro-architectural properties of the accelerators and the benchmark properties.

1 Introduction

With all the available transistors and the immense power dissipation of monolithic processor cores, much focus is placed on *accelerators* as a way to increase computational strength and efficiency of processors. These accelerators can be integrated on-die, as in STI's Cell processor [5] and the POD accelerator [9], or they may be realized in accelerator boards as in GPUs [4], ClearSpeed's CS301 [3] and Nallatech's Slipstream FPGA-based accelerator [1].

Accelerators are already a success story in application fields such as 3D-graphics, physics and encryption. As the number of accelerators and their processing power increases, other application fields can also be interested to take advantage of these resources. However, the question is whether applications from outside high performance computing can also benefit from the current accelerators?

2 Approach

2.1 Identification of sub-algorithms

A first problem to tackle is which parts of a program are suitable for offloading to accelerators. If the programmer has intimate knowledge of the program, finding the suitable partitions to offload is probably possible with some insight and intuition. In general, however, people are not familiar with all the intricate details of a program, so a more systematic approach is required. In recent work [6] a first step for finding suitable program partitions (so-called *sub-algorithms*) to offload to accelerators was introduced. The criteria are based on control and data flow characteristics.

Once the interesting regions to offload to an accelerator are identified, the work is not yet done. One has to transform and optimize the offloaded code to suit the targeted accelerator. In this paper we look at the performance improvement of different optimizations when using the SPUs of a Cell BE processor as accelerators for the main PPU-processor core.

2.2 Three accelerator contestants

We consider three different benchmarks for finding suitable sub-algorithms to offload on an accelerator. The first benchmark is *Clustal W* [7], a bio-informatics program used for the simultaneous alignment of many nucleotide or amino acid sequences. The program has three main stages of which the first, *pairwise alignment*, is an interesting sub-algorithm to offload. It is characterized by consuming a lot of data and is dominated by regular array-operations. Moreover, 99% of the execution time is spent in loop nests. These features makes it very suitable for streaming data and having a very predictable control flow.

The second program is *bzip2* from SPECINT2000, a program for compression and decompression. The compression takes about 86% of the execution time and spends about two thirds of its time on sorting. The interesting sub-algorithm in this case is *simpleSort*, which performs a shell's sort. It takes about 20% of the execution time. The shell's sort is called by the main sorting routing (*sortIt*) and by a quickSort algorithm (*qSort3*). Note that sorting algorithms have an unpredictable data behavior and unbiased branches. Furthermore, only small parts of the code are vectorizable due to the intensive control flow.

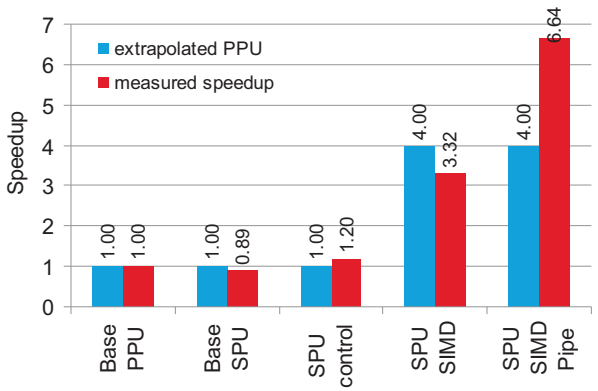

Figure 1. Speedup results for pairwise alignment of Clustal W

The third program, *mcf*, is also from SPECINT2000 and solves a large-scale minimum-cost flow problem. Three quarters of the time is spent in *primal_net_simplex*. The interesting sub-algorithm here is *primal_bea_mpp*, responsible for 41.3% of the execution time and consists of several nested loops and the function *sort_basket*. The network is represented as a graph, resulting in pointer-chasing and exposed memory latencies. Combined with control flow, there is no opportunity for vectorization. However, the loop structure allows to overlap some DMAs [2].

2.3 Extrapolated speedup from PPU

In the analysis (Section 3) we compare the measured speedup on the accelerators with the *extrapolated PPU* speedup. For the extrapolated speedup we take the fraction of vectorized code into account and the length of these vectors. Furthermore, we also use the number of parallel threads and fraction of parallel code. For the extrapolated speedup we assume no communication overhead and use the same memory latency as for one PPU.

3 Analysis

The benchmarks are compiled with gcc 4.1.2 and run on a PlayStation 3. The presented results are relative speedups compared to the performance on a single PPU-thread.

Clustal W When running the original version of *pairwise alignment* on a SPU we get a slow down of 10% (`Base SPU` in Figure 1). Removing control flow by using compare-and-select construction [8] gives a speed improvement of 20% (`SPU control`). The SPU is very sensitive to control flow because, as opposed to the PPU, it lacks a dynamic branch predictor. On the base processor this would not result in a measurable speedup.

Vectorization (`SPU SIMD`) could ideally lead to a speedup of a factor 4, but due to extra overhead we only get a factor of 2.75 compared to `SPU control`.

Table 1. Pipeline utilization, in percentage of execution time.

Benchmark	Clustal W	Bzip2	Mcf	
Kernel	Pairwise	SmpleSrt	prim_bea_mpp	
Usage	SPUx	SPUx	SPU0	SPUx
Single cyc	40.8	26.2	27.9	30.7
Dual cyc	50.7	4.2	2.1	1.6
Nop cyc	0.3	1.4	0.3	1.6
Br miss	0.5	11.8	10.7	14.9
Dep SPR	0.0	0.0	48.4	3.4
Dep other	6.8	10.7	8.4	29.0
Chan stall	0.9	37.6	2.0	18.1

Some of the memory accesses are unaligned, which severely deteriorates the performance on an SPU. By unrolling the loop (`SPU SIMD Pipe`) these unaligned accesses can be avoided, effectively reducing the instruction count of the loop. The same optimization has no expected effect on the PPU, as the PPU handles alignment in hardware. Overall, the SPU is able to execute Clustal W very efficiently, even better than what optimistic extrapolation suggests. This is also reflected in the pipeline utilization (Table 1): only a small fraction of cycles is spent on dependencies and channel stalls.

Bzip2 In Figure 2 we see that offloading *simpleSort* to one SPU is about two times slower than running the whole program on the PPU (`SimpleSort SIMD`). Different parts of the code can be vectorized to some extent, allowing to reduce the number of compares. Based on the reduction of compares, extrapolation would result in a speedup of 1.98 when vectorizing the code. In practice, however, we get a slow down of more than 2. Note that this result already incorporates an aligned data lay-out, minimized control flow and static branch hints. The overhead is caused by communicating messages and the delay of DMAs.

Luckily there is parallelism between the calls from *qSort3* to *simpleSort*, allowing to use several SPUs. This is shown in the bar `SimpleSort SIMD Par` of Figure 2. Using 6 SPUs speedups *simpleSort* with 56%. This results in a 14% speedup of the compression and a global speedup of 9%. The extrapolated speedup however, shows that the bzip2 program does have significant opportunity for acceleration, but control flow and communication delay means its not exploited. A larger fraction of branch misses and channel stalls in Table 1 confirms this. Note that applying static branch hints is not useful in this case, as these branches are unbiased.

The lower speedups for `Compress` and `Total` are just a consequence of Amdahl's law: since the sequential fraction is larger, the speedup is lower. The difference between extrapolated and measured speedup indicates we only achieve 50% of the potential speedup.

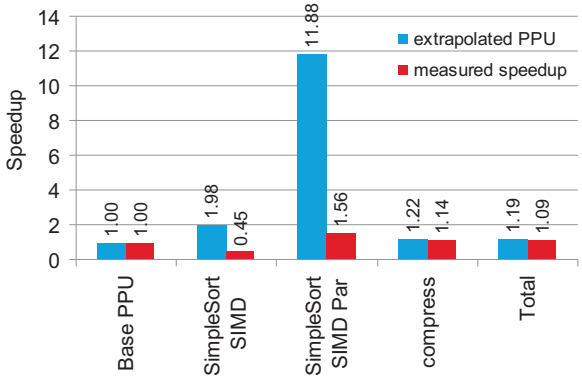

Figure 2. Speedup results for bzip2

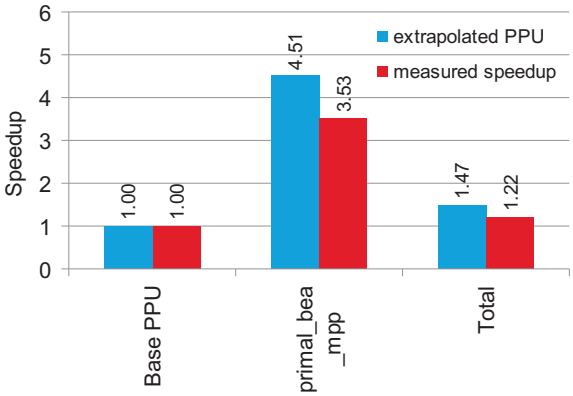

Figure 3. Speedup results for mcf

The reason for not offloading the complete *sortIt* procedure (which takes about 60% of the time) is that it is very control intensive and results in degraded performance due to the lack of dynamic branch prediction.

Mcf The execution of `primal_bea_mpp` can be parallelized over 6 cores. In Figure 3 we see that the measured speedup is lower than the extrapolated speedup (deduced from the available amount of parallel work). The measured speedup is about 3.5 times, after applying similar optimizations as for Clustal W (data alignment, avoiding control flow, ...). In this case the code is less suitable for execution on a SPU, since no useful vectorization is available.

The speedup is comparable to the extrapolated speedup because the loop structure allows to overlap quite some DMAs, but the total measured speedup is still only halve of the extrapolated. So again lack of vectorization potential and unbiased branches prevents the program from exploiting the full potential of the accelerator cores. In Table 1 we make a difference between one SPU that is used for distributing the work and the other ones (SPUx) performing the actual work. This first one spends about halve of its time on polling the SPU channels (Dep SPR). The other ones have mainly load/store dependency (Dep other) stalls.

4 Conclusion

The current accelerators, in particular the SPUs, are optimized towards application fields of number crunching. These fields are characterized by predictable control flow and data behavior suitable for streaming and vectorization.

However, by creating some more general accelerators many different programs could greatly benefit. Our case studies show that enabling control flow prediction or reducing DMA latency, would result in significant performance improvement for programs that have less regular control-flow and data behavior.

Acknowledgments

The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research. Sean Rul is supported by a grant from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Hans Vandierendonck is a post-doctoral research fellow with the Fund for Scientific Research-Flanders (FWO). This research is also funded by Ghent University and HiPEAC.

References

[1] A. Cantle and R. Bruce. An Introduction to the Nallatech Slipstream FSB-FPGA Accelerator Module for Intel Platforms. White paper, http://www.nallatech.com, Sept. 2007.

[2] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *Computing Frontiers*, pages 1–8, 2006.

[3] T. R. Halfhill. Floating point buoys ClearSpeed. *Microprocessor Report*, page 7, Nov. 2003.

[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[5] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, and et al. The design and implementation of a first-generation CELL processor. In *IEEE International Solid-State Circuits Conference*, pages 184–592, 2005.

[6] S. Rul, H. Vandierendonck, and K. D. Bosschere. Towards automatic program partitioning. In *Computing Frontiers*, pages 89–98, 2009.

[7] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res*, 22(22):4673–4680, 1994.

[8] H. Vandierendonck, S. Rul, M. Questier, and K. De Bosschere. Experiences with parallelizing a bio-informatics program on the Cell BE. In *3rd HiPEAC Conference*, pages 161–175, 2008.

[9] D. H. Woo, H.-H. S. Lee, J. B. Fryman, A. D. Knies, and M. Eng. POD: A 3D-Integrated Broad-Purpose Acceleration Layer. *IEEE Micro*, 28(4):28–40, 2008.