CPCPU: COREFUL PROGRAMMING ON THE CPU
Why a CPU can Benefit from Massive Multithreading

G. De Samblanx[1,2], Floris De Smedt[1,2], Lars Struyf[1], Sander Beckers[1], Joost Vennekens[1,2] and Toon Goedemé[1]

[1]*Campus De Nayer, Lessius, Belgium*

[2]*Department of Computing Science, K.U.Leuven, Leuven, Belgium*

Keywords: GPGPU, CUDA, OpenCL, Massive Multicore.

Abstract: In this short paper, we propose a refreshing approach to the duel between GPU and CPU: treat the CPU as if it were a GPU. We argue that the advantages of a massive parallel solution to a problem are twofold: there is the advantage of an excessive number of simple computing cores, but there is also the advantage in speed up by having a large number of threads. By treating the CPU as if it was a GPU, one might end in the best of two worlds: the combination of high performing cores, with the massive multithreading advantage. This approach supports the paradigm shift towards massive parallel design of all software, independent of the type of hardware that it is aimed for.

1 INTRODUCTION

By introducing the concept of a 'general purpose GPU',manufacturers of graphical processors promote the use of their hardware more as a replacement of a CPU than as a classical numerical coprocessor. Nevertheless, the CPU is still the core in a system that sources out the bulk of its work (Beckers, 2011).But maybe it is time to turn the table. In this text we explore the advantages of using a CPU in the role of a 'supporting' GPU.

Note: the experiments in this article were run on a *Intel i5-2400 Quadcore 3.1Ghz* and a *NVidia GTX480* graphical card.

2 THE POWER OF GPGPU

In recent years, a large interest arose around the use of graphical processing units (GPU) for general computing purposes. Advocates of GPU programming claim that the use of these 'massive' number of parallel working units will lead to a cheap but vast army of computational power. NVidia predicts for its Tesla hardware combined with CUDA software technology, a speed advantage factor of 100 by the year 2021 – that is 100×, not 100% speed up (Brookwood, 2010). ATI observes a more humble factor of 19× with current technology (Munshi, 2011). Using OpenCL, the ViennaCL benchmarks (Rudolf et al., 2011) report a speed up factor between 3 and 18, depending on the application, with minimal optimization. So the idea is: laying your hand on few teraflops at the cost of a netbook, who could refuse?

Even though the old school parallel computing community has been reluctant to admit it, massive parallelism is a different paradigm. The classical approach is to divide an algorithm in a sequential and a parallel part. Then the parallel part is executed on the multiprocessor. In contrast, massive parallel computing often requires a new algorithm design (Beckers, 2011).

But it also suffers from Amdahl's law. Simply stated, Amdahl's law says that the speed (up) of an algorithm is limited by its sequential fraction S, since only the parallel fraction $P = 1 - S$ can benefit from parallelization:

$$\text{speedup} = \frac{1}{S + \frac{P}{\#cores}}.$$

If the number of parallel cores becomes very large, as is the case in a GPU, the speedup levels off at approximately $1/S$. Therefore, it might be a good idea to use *faster* cores, instead of more cores, and jump to a higher performance curve, as is shown in Figure 1. Notice that if we would replace the 'number of cores' on the x-axis by a 'number of threads on a limited set of cores', then the results even get worse. Additional

DOI: 10.5220/0003904901960199
In *Proceedings of the 2nd International Conference on Pervasive Embedded Computing and Communication Systems* (PECCS-2012), pages 196-199
ISBN: 978-989-8565-00-6

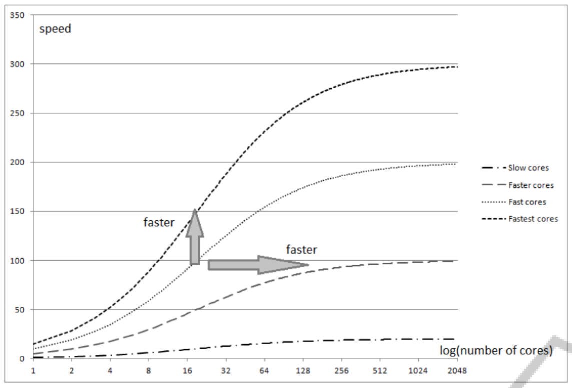


Figure 1: Amdahl's law: computational speed versus the number of cores; logarithmic scale with $S = 5\%$.

overhead discourages this approach mst strongly.

3 PEAK PERFORMANCE IS NOT A CONSTANT NUMBER

The real-life peak performance if a computer core is, unlike commercial statements, not a constant number. It depends on the size and the type of the problem that runs on the hardware. E.g. problems that are too small do not result in an optimally filled hardware pipeline, whereas large problems do not fit in local memory. A typical 'peak' performance curve as a function of the problem size has the staircase-pyramid shape as shown in Fig. 1. To the left of point A, the curve rises

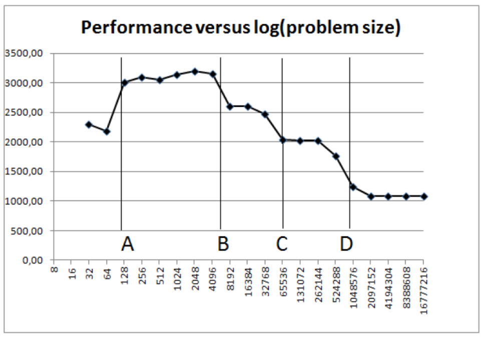
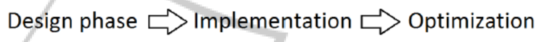


Figure 2: The evolution of peak performance as a function of the problem size (log scale).

while the pipline fills up and overhead decreases. At the right of point B, the performance drops when the different caches fail to store the data. In this example, the region B-C uncovers the Level1 cache, whereas C-D shows the L2 cache. The curve was drawn by running repeatedly the very simple BLAS1 routine `snorm2` on one CPU core (Duff et al., 2002). This is a routine with almost no overhead.

The problem with this curve is that the region A-B of maximal performance is also the region where this performance is not really needed: the problem is too small. In practice, this performance will probably be hidden behind the overhead of setting up the next, small problem. For very large data, beyond point D, on the other hand, the performance is at the level of really cheap hardware.

Moreover, performance can only be achieved by optimization. Optimization implies the mapping of an implemented solution on specific hardware; in some cases, it can make the porting of software to different hardware awkward. In practice, one starts by mak-



ing a sequential design that is easier to debug. Then, as many parts of the design as possible are implemented in parallel. after which they are optimized and mapped on the hardware.. If the software is ported from/to CPU/GPU afterwards, then the optimization must be redone in a very fundamental way.Sometimes the whole algorithm must be thought over again. With this discussion paper, we propose a different approach: avoid the choice between CPU/GPU, but try to design the software solution from the start within a massive parallel paradigm. As we will argue in the following paragraph, this is not necessarily bad for the CPU. In fact, the CPU may even benefit from being treated like a GPU.

4 CPU MASSIVE MULTITHREADING

Let us review the graph of Amdahl's law. It is easy to see that the 'number of cores/threads' on the x-asis of Figure 1 are related directly to the problem size-per-thread:

$$\text{size per thread} = \frac{\text{total problem size}}{\text{number of threads}} + \text{overhead}.$$

By ignoring the overhead, we can combine Fig. 1 and Fig. 2. The points A-D appear in reverse order on the combined graph, since the problem size is related to the inverse of the number of cores. Becaouse these points indicate where the performance of one core jumps to a higher/lower curve, they indicate the points where an application jumps to a faster/slower Ahmdal curve, as is shown in Figure 3. The good news here is that the A-B region of maximal performance, is located much more to the right of the curve (where the problem size is large). So with a *fixed*

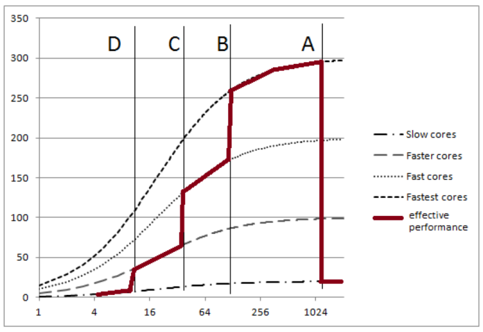


Figure 3: Combination of Amdahl's law with single-core performance; performance speed versus log(number of cores).

problem size per thread, the implementation will run faster for larger problems. On a GPU, the problem size per thread is not fixed. It depends on the number of cores versus the overall problem size, which are both fixed. But on a CPU with e.g. 4 cores, the number of threads can vary in a continuous fashion to any number, changing the problem-size-per-thread at will.

Figure 4 shows the results of this experiment on our test system. We ran the same problem from Figure 1 using 1 to 480 threads on a quadcore Intel i5. Since every core runs at about 3GHz, a maximal performance of 12GFlops can be expected – assuming 1 flop per clock cycle. Obviously, the performance doubles when going from 1 thread to 2 and quadruples for four threads. But all of those four implementations level off around the same point at approximately 1GFlops. In other words, for problems larger that 2 megabytes, the four cores *combined* perform as poorly as only one. Now, if we run the same problem with 48 to 480 threads on the 4 CPU cores, then the region of maximal performance clearly shifts to the right. This 'optimal' region also tends to be much

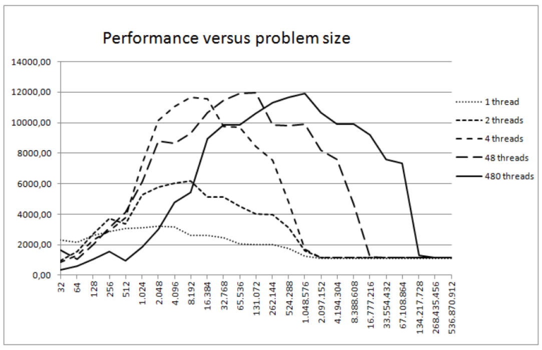


Figure 4: Running `snorm2` with 1 to 480 threads, on 4 cores for increasing problem size (logarithmic scale).

broader, remember that the x-axis is labeled logarithmically. Very little performance is lost in the extra overhead of launching the threads.

Of course, this level of optimisation could also be achieved by classical optimization and by probing the speed of the internal busses and the size of the caches. But by massive multithreading, the number of threads is an optimization parameter that is quite independent of the hardware. This makes the porting and the reuse of software easier.

5 CONCLUSIONS

With the advent of massively multicore GPUs came the new programming paradigm of massively parallel computing. In this paper, we have argued that the usefulness of this new paradigm need not be limited to the GPU: there are sound theoretical reasons to expect that the same programming style may also be beneficial for programs run on CPUs. In particular, it may helps to shift the performance curve towards the end of the spectrum where performance is most needed, i.e., the large instances. At the negligible cost of solving the small instance slightly less quickly, the number of instances that can actually be solved can be drastically increased, as demonstrated by the preliminary experiment discussed in this paper.

A second advantage is of course that, by using the same programming paradigm on both CPU and GPU, the choice between those different types of hardware can be postponed. A suitably designed algorithm may be deployed on either one with only minimal changes. Moreover, if a platform-independent parallel programming language such as OpenCL is used, it may not even be necessary to change the implementation either.

It is not necessary to know at design time how many cores the hardware will have, since the number of threads (not cores) is a possible optimization parameter. While executing these threads, CPUs can address a larger amount of memory than GPUs and have still more efficient pipelines. On the other hand, the CPU cores are definitely much lower in number.

Finally, debugging software on a CPU platform is much better supported than on a GPU. So if designed software can be transferred between CPU and GPU as if they were highly similar, then the debugging problem of GPUs is somewhat relieved.

ACKNOWLEDGEMENTS

Parts of this research is funded by the IWT project *"S.O.S. OpenCL - Multicore cooking"*.

REFERENCES

Beckers, S. (2011). Parallel sat-solving with opencl. In *Proceedings of the IADIS Applied Computing*.

Brookwood, N. (2010). Nvidia Tesla GPU computing, revolutionizing high performance computing.

Duff, I. S., Heroux, M. A., and Pozo, R. (2002). An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28:239–267.

Munshi, A. (2011). *OpenCL and heterogeneous computing reaches a new level*. AMD DeveloperCentral.

Rudolf, F., Rupp, K., and Weinbub, J. e. a. (2011). *ViennaCL User Manual*. Institute for Microelectronics, TU Wien.