On Programming Models for Multi-Core Computers

Frank Feinbube
Frank.Feinbube@hpi.uni-potsdam.de

This document gives an overview of my current view on the topic of my promotion and the current state of the research I have done since May 2009 at the HPI Research School on Service-Oriented Systems Engineering within the operating systems and middleware group supervised by Prof. Dr. Andreas Polze.

Since services are hosted by application servers, shifts in the underlying systems have a great influence on their efficiency and functionality. Therefore it is necessary to get a deep understanding of trends in operating systems and middleware as well as hardware environments. This paper discusses some of these shifts. It will take a short look on energy efficiency and a deeper one on programming models for multi-core computers. Thereby first the application of graphic cards for parallel computing will be evaluated using the example of CUDA. Then a brief summary on the limits of threads for the Windows platform is shown.

1 Introduction

Services, especially web services, are hosted by application servers running in enterprise server environments. To create and maintain services that are reliable and deliver a good performance, one has to understand the underlying layers. Not only application servers have to be considered, but also trends in operating systems and hardware. Knowing the whole system provides the foundation to optimize every level for the execution of services. In addition this is the only way to generate useful service meta-data information such as the reliability, scalability and carbon footprint of a service.

One of the most popular trends is the architectural shift towards multiple processing units. It leads to a situation where software can only benefit from Moore's law, if it can be parallelized in a way that exploits that new architecture. In other words: if we can not adapt, new software will be slower on emerging processor architectures. The reason for this shift is the fact that increasing computational power and density of transistors not only results in increased power requirements, but also in additional heat production. For every 10 °C increase in temperature the failure rate of a system doubles (Arrenhius' equation) [8]. This is why cooling and power supply have become the most expensive parts in high-performance computer centers. Two ways lead out of this misery. On the one hand we have to consider energy savings to decrease the costs of maintaining computer systems. This topic is discussed in section 2. On the other hand we have to write software to be highly parallelizable to benefit from new architectures. This

way computation can be distributed and thus heat issues can be managed with greater efficiency. Unfortunately writing parallel programs is not as easy as using a new API or learning a new programming language. Actually we have to learn a new way of thinking about the software problems we face and the way we solve them. Section 3 is devoted to the subject of programming models for multi-core environments.

Another development, which rises straight out of its cradle, is cloud computing. The greatest change is the shift of responsibility for the availability and administrative work to the hands of an external provider. This allows users to concentrate on their core business. The cloud will be highly scalable and users will only have to pay for the compute power they use. Just as we now pay for electricity from the socket. Currently, many details are still in the dark. Thus, it is unclear what kinds of software systems will benefit from these platforms and how these will be written. But one thing is for sure: the underlying program must be highly scalable. In the context of this paper I will not consider cloud computing in detail.

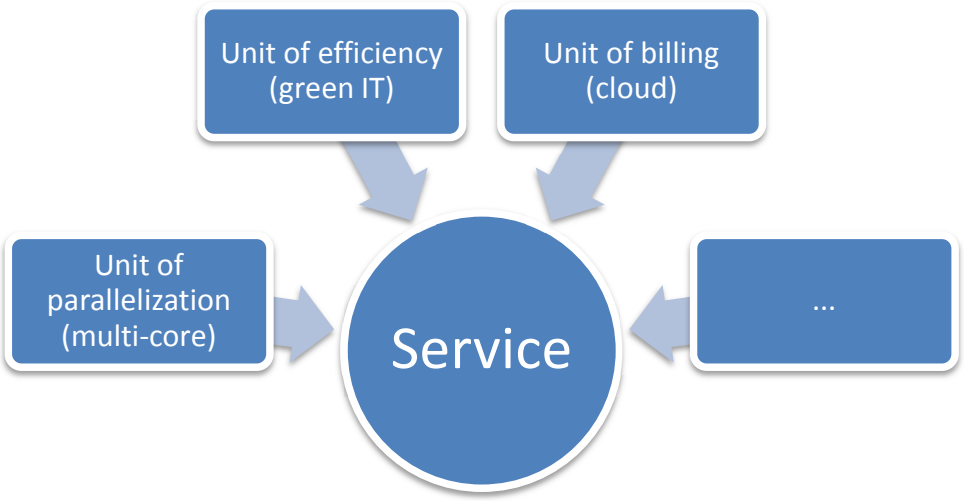


Figure 1: Services are the unit of scale in different domains.

As illustrated in Figure 1 all these shifts support the idea of seeing software as a service. But they also suggest that we should revise our service notion. This way we can check if it can handle the new developments in a good fashion and see spots where we can adapt it to derive the greatest benefits from the new situation.

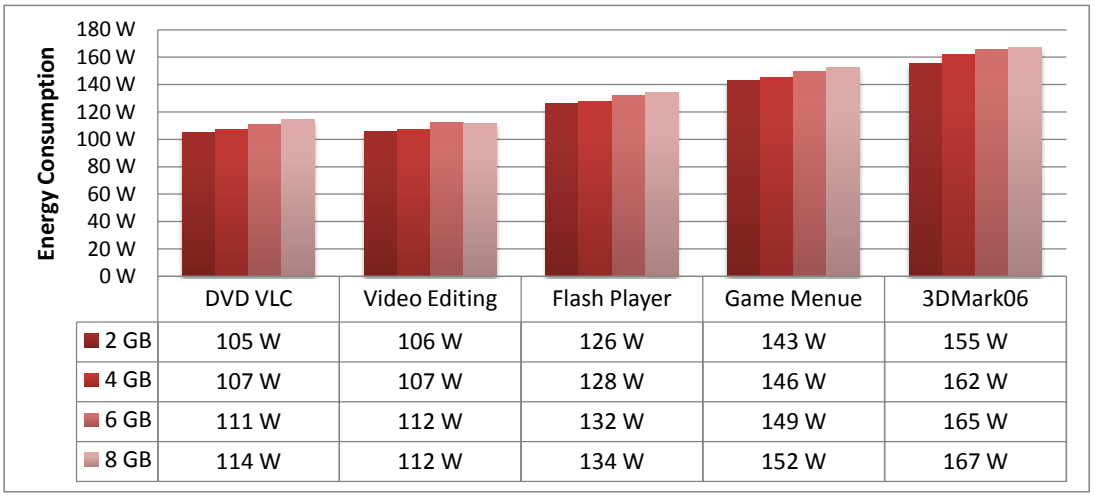
2 Energy Efficiency

Many Application servers are hosted in huge computer centers. In order to fulfill their high performance requirements these centers make use of a lot of computers. In the past the hardware of these computers was the main expense in building and maintaining a high performance computing center. But this is no longer true. Nowadays the costs for power supply and cooling of the hardware are about 50 to 70% [4] of the overall costs of the computer center.

In such a situation it is import to find new ways to reduce the energy consumption and the heat generation of the applied hardware components. This reduction should be accelerated by means of the operating system which should orchestrate the hardware components in an intelligent way. [8] and [22] show that such an approach can save a big amount of energy for a very small performance loss. In [18] a solution for CPUs is presented that not only reduces the energy footprint of the computer, but also increases the resulting performance.

Looking at the energy problem from the viewpoint of services we see a lack of convenient metrics. Having meta data information on the energy consumption and the CO_2 footprint of a service would make it possible to compose them in a more intelligent fashion. This way we could not only predict the energy needed to use a service, but also compose it in a way that minimizes the CO_2 footprint. Services are small, composeable and annotated with meta data by design. Thus they provide a solid basis for considerations on energy efficiency.

As described in [3] in the future power-aware features should be available to all major parts of the system. While there are big improvements in CPU and hard disk architectures, I found little research on the field of main memory. Thus I evaluated the energy consumption of main memory bricks for different usage scenarios. Therefore a test system was configured with different amounts of main memory. In all configurations the efficiency in handling the workload as well the energy consumption was measured.



	DVD VLC	Video Editing	Flash Player	Game Menue	3DMark06
2 GB	105 W	106 W	126 W	143 W	155 W
4 GB	107 W	107 W	128 W	146 W	162 W
6 GB	111 W	112 W	132 W	149 W	165 W
8 GB	114 W	112 W	134 W	152 W	167 W

Figure 2: Energy consumption of selected applications in relation to main memory brick count. A higher memory size leads to a bigger energy footprint, even if the additional memory is not used.

As depicted in Figure 2 the experiment showed that each memory brick adds about 2% to the memory footprint of the system. To make it worse this is the case even if the memory is not used at all. If this holds for most desktop computers it would be desirable to create hardware and operating system support to disable unnecessary memory bricks dynamically. This way energy savings could be brought one step further without any reduction in quality or service penalties. This technique would lead to approximately 6% power savings in the test system.

In order to get a broader view on the problem it is important to test some other desktop computer configurations as well as server computers. In addition a comparison of different memory brick sizes and vendors would be interesting.

3 Programming Models for Parallel Computing

Finding a good way to write programs for multi-core environments has again become a very popular topic. While many new languages arise, most of them reuse concepts that were already known, for example in *communicating sequential processes* [7] and its implementation *Occam*. With *Intel's Threading Building Blocks* [9] and *Microsoft's Parallel Extensions to .NET Framework* [13] APIs are provided that aim to ease the use of multi-threading in *C++* and the *.NET Framework*. Amongst others they bring constructs that allow programmers to declare that loops are to be executed in parallel. Another parallel programming model uses agents that communicate via channels. *Microsoft's Axum* is based on such a model.

Lots of influence arises from fields where parallel computing is long since applied. One example is the area of distributed computing, where parts of the program are running on different machines. Here we can find fast accessible memory on the local machine, but a fairly slow communication channel to the server or peers. Similar problems show up with the *Non-Uniform Memory Access* (NUMA) in modern processor architectures.

Another example for fields that already have highly parallelized programs is the area of graphic processing units (GPU). In order to execute their particular tasks, graphic cards benefit from calculating many equivalent functions in parallel. Programming for graphic cards has long since been a very specialized task. The applied programming models were only a reflection of the hardware of these devices. But in order to exploit the cards for new visual effects there has been a shift to general purpose graphic cards (GPGPUs) for some years. Though this shift has not finished yet, graphic cards are becoming more general and CPUs are becoming more parallel. So perhaps they will converge at some point in the future. In subsection 3.1 I present my experiences with the *NVIDIA CUDA* programming model as one representative for GPGPUs.

Besides programming models, another problem that we have to face when we think about parallel computing are the restrictions of hardware and the limits of the operating systems and middlewares we use. I took a first step in studying the limits of thread creations which is presented in subsection 3.2.

3.1 General purpose graphic processing using CUDA

For some years graphic cards were not only used to render pictures to screens, but also for mathematical processing. Therefore shader languages or vendor specific languages like *Brook+*, *Cal* or *Cg* were applied. Today with languages like *NVIDIA CUDA* [15] and the *AMD Stream Computing SDK* [1] it is even possible to write programs using only a few extension to the *C* programming language. The next step will be the application of the emerging *OpenCL* [2, 10] API. It will allow to program algorithms

that abstract from CPU and GPU as the underlying processing device. Current OpenCL implementations are not yet fully usable. *AMD* provides an implementation that is only capable to use the CPU while *NVIDIA* only supports the use of their CUDA-enabled graphic cards. This is why this section focuses on CUDA which is a well established "scalable parallel programming model and a software environment for parallel computing." [19]

It allows writing programs that run on general purpose graphic processors (GPGPU) by NVIDIA. Due to the hardware architecture of these devices, many complex computational problems can be solved much faster than on current CPUs. These problems include physical computations and video processing. Nowadays development for CUDA is done using some C extensions. The code is precompiled with a compiler provided by NVIDIA and finally compiled to binaries accessing CUDA-enabled graphic drivers. For CUDA works with all modern graphic cards from NVIDIA, even *NVIDIA ION* [14], it is available in hundreds of thousands of computers. This makes it particularly interesting for research on parallel computing.

3.1.1 Programming Model

The design goals of the CUDA programming model are to enable programmers to develop parallel algorithms that scale to hundreds of cores using thousands of threads. [19] Thereby the developers should not need to think about the mechanics of a parallel programming language and should be enabled to employ the CPU as well as the GPU.

The application parts that are executed on the graphic card are called kernels. They are executed by lots of lightweight CUDA threads which can communicate using slow device memory or fast shared memory. While the kernel is running the CPU is free to handle other workload. Each kernel has some equivalent tasks to fulfill. Listing 1 shows such a typical kernel execution scheme. At first each thread can calculate its unique thread identifier using some constructs provided by the CUDA environment. This identifier can be used to make control decisions and to compute the memory addresses of the input parameters. Finally the calculated result is written back to the global memory.

```
// derive absolute thread id from block id and relative thread id
int threadId = blockIdx.x * blockDim.x + threadIdx.x;
int parameter = dataGpu[threadId];}   // read from array (using id as index)
result = parameter * parameter;}      // calculate the result
dataGpu[threadId] = result;}          // write to array (using id as index)
```

Listing 1: CUDA kernel execution scheme

There is a strict separation of kernel routines and normal program routines. Kernel code cannot access host memory. Kernel methods must not be recursive and must not use static variables.

3.1.2 Evaluation with an example

In order to get a deep understanding of the programming model, I chose an active research problem that is complex enough to demonstrate the abilities and limits of

CUDA. The *n queens puzzle* fulfilled these requirements. Its goal is to find all ways to place a given number *n* of queens on a chessboard which has *n* times *n* fields. A configuration is only valid if no queen attacks another one. This holds if no queen is placed in a row, column or diagonal that is used by another queen.

All solutions for this puzzle are known up to a number of 26 queens. Preußer et al. from the University of Dresden [5] calculated all 22,317,699,616,364,044 valid configurations using specialized FPGA boards. The competitor to this project is *NQueens@Home* by Figueroa from the Universidad de Concepción de Chile [6] who uses an approach that is similar to *Folding@home* [17] where a middleware is provided that distributes work packages over the Internet. While calculations are done by the personal computers of registered users in both cases, NQueens@Home does not exploit their graphic cards. This is critical, because the statistics of the Folding@home project [16] show, that the application of graphic cards brings a huge performance benefit compared to common CPUs.

Both projects are based on an optimized single-threaded algorithm written by J. Somers [23]. After investigating the problem and programming an own solution, I decided to take the algorithm of J. Somers as the basis for my CUDA version as well. This decision founded on the fact that this algorithm does not use recursions, consumes little memory and is widely accepted.

The first and most important step was to parallelize the algorithm. Therefore I modified it in a way that allowed the precalculation of board settings for a given number of rows. Its results are then used as the input for an algorithm that calculates all solutions starting from the given setting. Applied to CUDA, the first algorithm runs on the host and the second one as a kernel on the graphic card. After the basic program was running, I step by step modified it to use fast shared memory instead of slow mapped device memory for the arrays. As shared memory is rare on graphic cards, using more of it reduced the number of threads I could deploy.

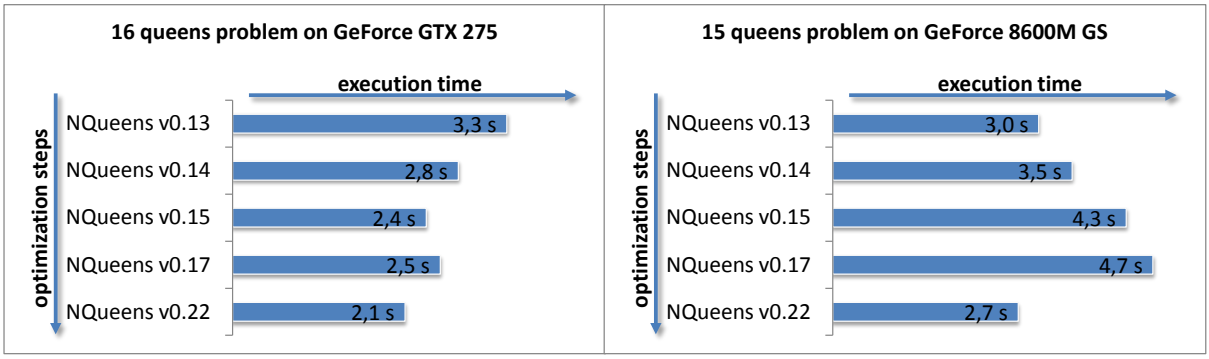


Figure 3: Runtime comparison of different versions of the NQueens program. The higher the version number the more shared memory is used instead of device memory. While the optimizations lead to better performance on the latest graphic cards (Geforce GTX 275), it results in worse performance on former CUDA-enabled versions (Geforce 8600M GS).

In Figure 3 the performance impact of these optimization steps is shown. Using a graphic card with the latest CUDA-enabled architecture from NVIDIA (*Geforce GTX 275*), the shift from device to shared memory lead to a performance increase. The benefit from faster memory accesses exceeded the penalty of the reduced thread count. On the other hand the same optimizations lead to a decreased performance on the former architecture (*Geforce 8600M GS*). Surprisingly after the last optimization step the performance is significantly improved and outperforms the original program version. In the last version shared memory is used for all arrays and the memory footprint per thread is minimized as well. The reason for the different results of the optimizations on the different architectures is a performance problem that arises from incoherent memory accesses on later CUDA-enabled architectures.

This evaluation shows that general purpose graphic cards are still far away from CPUs, because programmers can not even make assumptions on the effects that optimizations of a program will have. A program that is optimized for one architecture will not necessarily run fast on another CUDA-enabled architecture.

3.1.3 Conclusion

During my experience with the CUDA programming model I learned that there are a lot of restrictions a programmer must cope with. At first there is the unavailability of recursion and the differentiation of CUDA subroutines from normal routines. This is not only very unfamiliar to C programmers, but leads to replicated code as well. The second one is the focus on minimizing the memory usage of the algorithms. As a programmer one has to think "more carefully about memory access patterns and data structures sizes". [11] Another problem with memory that I faced was the unpredictability of the resulting register count. I was not able to derive a pattern for the number of occupied registers from code changes. Unfortunately one needs to know the number of occupied registers at coding time to optimize the code accordingly.

The last and most surprising restricting is that CUDA threads are only allowed to run for a very short period of time. Long running CUDA kernels result in a driver restart will thus be canceled. If a program needs to run several seconds or more, the display mode of the operating system must either be deactivated or additional graphics hardware must be used to take over the rendering work.

There is still a lot of work to do until general purpose programming for graphic cards will be as comfortable as programming for CPUs.

3.2 Pushing the Limits: Processes and Threads

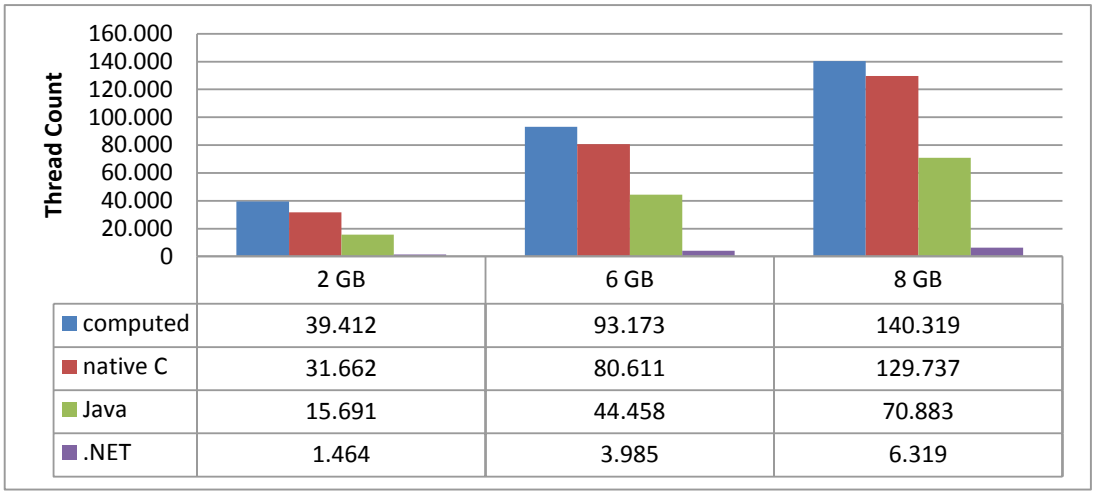
The usage of multiple processes and threads as workers is a widespread programming model. Inspired by an article of Mark Russinovich [20] and in cooperation with a vendor of complex enterprise applications, we tried to figure out what are the limits in the usage of processes and threads. The first approach which is described in this section was the creation of a model to predict the number of threads that can be created on a 64-bit Windows operating system. While lots of information is present for 32-bit [20], there is little information available for a 64-bit Windows operating system.

After studying the Windows internals to figure out how much memory a thread consumes [21], we came up with a fairly simple model:

$$max\_threads = \frac{(available\_memory + available\_pagefile\_memory) - program\_heap}{kernel\_thread\_size + thread\_commited\_memory}$$

$$kernel\_thread\_size = size(kernel\_stack) + size(TEB) + size(ETHREAD)$$

The maximal thread count equals the amount of memory that is available for thread stacks divided by the size of a thread. A thread's size is the sum of its user mode as well as its kernel mode representation. Due to some additional kernel constructs the size of kernel representation of a thread is negligible bigger than in our model. In order to check the accuracy of this model, we compared it against some benchmark results.



	2 GB	6 GB	8 GB
computed	39.412	93.173	140.319
native C	31.662	80.611	129.737
Java	15.691	44.458	70.883
.NET	1.464	3.985	6.319

Figure 4: Maximal thread count that can be achieved for different memory sizes on a 64-bit Windows 7 operating system using C, .NET and Java compared to a computed value based on our model.

Figure 4 shows the results of our evaluation. We found that our model fits very well for native C. The divergence is due to *Address Space Layout Randomization* (ASLR) which is included in Windows operation systems since *Windows Vista*. Using the Java environment half of this thread count can be created. That penalty is due to the overhead for the thread management of the Java Virtual Machine. With the .NET framework the possible thread count is even smaller. This is due to the fact that .NET relies on committed memory which potentially wastes resources for they are unused. Further inspection of the *Shared Source Common Language Infrastructure* (SSCLI) [12] would be necessary to check what improvements are possible here.

While studying the Windows internals and investigating the limiting factors for thread creations within the operating system, a lot of interesting questions arose that we will try to answer in the future.

4 Conclusion

This paper gives an overview of current architectural shifts and their impact on service computing. In section 2 energy efficiency considerations are presented. The need for operating system and middleware support to reduce the power consumption of a system by deactivating unnecessary system parts is illustrated by the example of memory bricks. Furthermore in section 3 the emerging importance of programming models for multi-core computers are discussed. Taking CUDA as one representative, general purpose GPU programming models were studied. The restrictions of these models were shown by evaluating a solution to the n queens problem. Some of the limits and problems of parallel computing were examined.

As shown in this paper we face a lot of interesting architectural shifts today. In order to let service computing benefit the most from these shifts, we need to get a better understanding of them and to revise our service notion appropriately.

References

[1] Advanced Micro Devices. ATI Stream Software Development Kit (SDK). http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx, 2009.

[2] Apple. Apple - Mac OS X - New technologies in Snow Leopard. http://www.apple.com/macosx/technology/, 2009.

[3] Wu chun Feng, Xizhou Feng, and Rong Ce. Green Supercomputing Comes of Age. *IT Professional*, 10(1):17–23, January 2008.

[4] Dell. DELL Leitfaden für Stromversorgung & Kühlung. http://www.dell.com/downloads/global/solutions/PowerAndCooling_Brochure_EMEA-de.pdf, 2008.

[5] Thomas B. Preußer, Bernd Nägel, and Rainer G. Spallek. Queens@tud. http://queens.inf.tu-dresden.de/?l=en&n=0, 2009.

[6] Israel Figueroa. Nqueens@home. http://nqueens.ing.udec.cl/.

[7] C. A. R. Hoare. *Communicating Sequential Processes (CSP)*. Prentice Hall, 2004.

[8] Chung-hsing Hsu and Wu-chun Feng. A power-aware run-time system for high-performance computing. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 1, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Intel. Intel® Threading Building Blocks. http://www.threadingbuildingblocks.org/, 2009.

[10] Khronos. Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, 2009.

[11] William Mark. Future graphics architectures. *Queue*, 6(2):54–64, 2008.

[12] Microsoft. Shared source common language infrastructure. http://www.microsoft.com/downloads/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D&displaylang=en, March 2006.

[13] Microsoft. Parallel computing developer center. http://msdn.microsoft.com/en-us/concurrency/default.aspx, 2009.

[14] NVIDIA. Nvidia cuda-enabled products. http://www.nvidia.com/object/cuda_learn_products.html, 2009.

[15] NVIDIA. *NVIDIA CUDA™- Programming Guide - Version 2.3.1*. NVIDIA Corporation, 2.3.1 edition, August 2009.

[16] Vijay Pande. Folding@home - client statistics by os. http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats.

[17] Vijay Pande. Folding@home - distributed computing. http://folding.stanford.edu/.

[18] J. Richling, J. H. Schönherr, G. Mühl, and M. Werner. Towards energy-aware multi-core scheduling. *PIK - Praxis der Informationsverarbeitung und Kommunikation*, 32:88–95, 2009.

[19] Greg Ruetsch and Brent Oster. Getting started with cuda, 2008.

[20] Mark Russinovich. Pushing the limits of windows: Processes and threads. http://blogs.technet.com/markrussinovich/archive/2009/07/07/3261309.aspx, July 2009.

[21] Michael Schöbel. NtCreateThread: memory allocations in kernel mode. http://www.dcl.hpi.uni-potsdam.de/research/WRK/?p=86, 10 2009.

[22] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for os-level power management. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 289–302, New York, NY, USA, 2009. ACM.

[23] Jeff Somers. The n queens problem - a study in optimization. http://jsomers.com/nqueen_demo/nqueens.html.