

Lack of abstraction considered harmful

A review of the current state of programming on the Cell architecture

Chris Bunch

cgb@cs.ucsb.edu

Nelson Ijih

ijih@ece.ucsb.edu

ABSTRACT:

Today's programmers now have commodity parallel computing systems. Of them, we explore Sony's Cell Broadband Engine Architecture. The PlayStation 3, sporting a Cell processor, makes it easy to install Linux and explore the architecture. We review three papers on Cell development and add our experiences to it. We aim to convince the reader of the massive potential of the Cell architecture and the difficulties in using Cell due to the abhorrent lack of abstraction available. Although the architecture has been exposed to the user in the name of performance gains, we have seen it result in code that relies heavily on its version of the architecture, down to the amount of loops manually unrolled, to gain performance. This makes the code difficult to read, difficult to read, and incredibly difficult to program. Thus, we have named this paper in a similar style to an article online [6] that beautifully captures this point, itself named after the famous Dijkstra paper.

AN IMPROVED PROGRAMMING MODEL:

The authors present a new programming model for Cell, aptly named 'Cell superscalar' (CellSs), which automatically exploits the parallelism of a sequential

program through the different processing elements (SPEs) of the Cell B.E. Architecture. This model's focus is on how a simple and flexible a programming model for Cell would ease the programmer's workload. They propose a model composed of a source-to-source compiler and a runtime library. According to [1], CellSs allows the programmer to write sequential applications, and the framework is able to use the different components of the Cell B.E. (PPE and SPEs) by means of an automatic parallelization at runtime. This model is similar to the RapidMind framework we have used in programming some of the applications, in the sense that they both required the user to use annotations before the declaration of some of the functions used in the application, indicating which pieces of code will be executed in the SPEs. However, we noticed some discrepancies in using automated annotations at runtime vs. manually parallelizing the application code before compiling. There are cost issues involved in both, namely, the time for manual parallelizing, and or the cost that maybe incurred if and when the automated annotations fails to accurately put annotations in the code at runtime. Hence, the main point here is to see how intelligent the automated annotation framework is, and if it is able to perform its job efficiently. A side note from [1] is that the annotation before a function does not indicate that this is a parallel region. It just indicates that it is a function that can be run in the SPE. However, according to [1], the CellSs runtime builds a data dependency graph where each node represents an instance of an annotated function and edges between nodes denote data dependencies. From this graph, the runtime is able to schedule independent nodes to different SPEs at the same time. We observed that the effectiveness of this depends entirely on how many data dependencies exist (and thus how parallelism exists) between the various parts of the code.

The difference between this model and the others is that annotations that are specified are functions that are possible candidates to be run in parallel, but with the RapidMind framework, each instance of the programmer's annotated functions will all run in parallel. In the former, the data dependencies will determine which functions will be called in parallel. An example of the latter is our work (described later), where we run a matrix multiplication program on all the SPEs in parallel. RapidMind sees the annotation and will use it in parallel, although if there is another function after it that can be run in parallel, it may need to wait until the first function finishes (like in CellSs).

The corresponding data transfers are done by the runtime using the DMA engines. The call to the runtime is non-blocking and therefore, if the task is not ready or all the SPEs are busy, the system will continue with the execution of the main program. We see that calling an external runtime library would create additional workload on the PPE, and since the SPEs faster than the PPE, there may be scenario where the PPE would have to stall communication with other SPE's while dealing with the runtime library.

According to [1], the process of generating the annotated code involves three processes. First, calls to CellSs runtime initializing and finalizing functions are analyzed. Next, calls for registering the annotated functions are reviewed by the compiler, generating a table of functions that can be later indexed by the SPE generated code. Finally, the compiler substitutes the original calls to annotated functions by calls to the 'Execute' function from the CellSs runtime. Our observation of this was that even though RapidMind works the same way, it does seem to provide a more parallelized code, with just extra time spent by a programmer using it to manually put labels where a function or segment of code should be parallelized.

After compiling the code, the CellSs runtime analyzes each call to the ‘Execute’ function and manages the dependency tree. First, the called task is added as a node in the task graph. Next, a data dependency analysis of the new task with other previously called tasks is performed. The data dependency analysis is based on the assumption that two parameters are the same if they have the same address. The system also looks for data dependencies (RAW, WAR, WAR). It also handles the renaming of the input and output parameters. This is done in a similar fashion as register renaming is performed.

The programming model presented in this paper shows speedups with using their model, yet most of the models that we have reviewed tend to be more favorable to certain applications. The model does show it may be too early to label a specific model as the best to adapt in order to program Cell with ease and flexibility, but they all share a view of the potential computational capability of Cell. We believe merging or finding a common ground on our reviewed model would be a possible area of extension.

MULTIGRAIN PARALLELIZATION:

Since Cell B.E. has been in the commercial world for less than a decade, several researchers from the parallel computing community have been exploring ways to reduce the difficulty of its programmability while making sure it is not under utilized in computations that benefit from its heterogeneous nature. Yet another exploration of the Cell B.E. has been done by [2], but on a different perspective from the others. They experimented with a multigrain parallelization because of Cell B.E.’s uniqueness as a processor, thus, in this paper; they exploited the orthogonal dimensions of task and data parallelism on a single chip.

Some of the computational review we attempted on Cell was partially based on the functional and data decompositions techniques proposed in this paper. In [2], they proposed that the static or dynamic functional and data decompositions should be orchestrated carefully to fully utilize the SPEs, as we experienced in some of our computational models, some SPEs were under utilized while some were overloaded with computations. According to [2], implementing a dynamic scheduler with defined policies will allow applications to exploit the proper layers and degrees of parallelism, in order to maximize efficiency on the Cell's computational cores, including both the SPEs and the PPE. Using proposed policies from [2] was somewhat not as straightforward as written in the paper. However, their findings show convincing results as we learned the following from their proposal:

(a) The runtime system needs to be adaptive by choosing the form and degree of parallelism to expose to the hardware in reactions to workload variations. They claim that the right choice of forms and degrees of parallelism depend significantly on the workload and on the input, reducing the burden on programmers from manual unloads and offloads of data and computations between the PPE and SPEs. From previous Cell B.E. explorations, [2] has seen that static parallelization schemes failed to provide maximum performance on Cell. They were not able to achieve a high efficiency on the SPE's and aim to maximize Cell's performance in the future.

(b) With the event-driven multithreading execution engine presented in this paper, [2], they claimed that overloading the PPE yields a higher efficiency on the SPEs. They use a guided feedback scheduling policy for dynamically triggering and throttling loop-level parallelism across SPEs on Cell. They showed that work sharing of divisible tasks

across SPEs should be used when the event-driven multithreading engine of the PPE leaves half or more of the SPEs idle. This is particularly true for the case when there are no intense computations for Cell. In experimenting with RapidMind framework, we noticed differences following IBM's programmer's manual vs. putting annotations as instructed by the RapidMind developer's resource. It took longer when we failed to correctly annotate the code segments that should be parallelized. Therefore, their concept of a dynamic triggering of putting idle SPEs in use may indeed create an added advantage of fully utilizing Cell. However, this may create overheads on a systems level in situations where this scheduling policy wakes up idle SPEs and the cost of triggering an idle SPE is over the cost and time of waiting for the current SPE in operation to be used for the same computations. Perhaps this could be an extension on this paper.

In details to the above contributions made by [2], they observed that there are benefits from loop-level parallelization of off-loaded tasks across SPEs. However, we also observe that loop-level parallelism should be exposed only in conjunction with low degree task-level parallelism. Its effect diminishes as the degree of task-level parallelism in the application increases at runtime. The paper goes in depth by implementing a message-passing interface protocol (MPI). Then, they map the MPI processes on each thread on the PPE Cell. Since the PPE is a dual threaded engine [2], MPI processes on the PPE can utilize two out of the eight SPEs via concurrent function off-loading. However, we noticed that this might result in overloading an SPE, which may conflict with the equal amount of fair share on computations. This shortcoming could possibly be eliminated by a dynamic policy that determines at any given point in time, the scheme determine and assigned the number of SPE that would be dedicated to performing this

task. The second is a model for event-driven task-level parallelism (EDTLP), in which the PPE scheduler oversubscribes the PPE with more than two MPI processes, to increase the availability of computation for the SPEs [2]; however, the paper fails to mention the consequences of oversubscribing the PPE. We believe from our experience that this schema may not be suitable for certain scientific computations.

Lastly, evaluating Cell and comparing it with other processors is always the ideal comparisons to proving the potential of Cell on scientific applications. In their comparisons, results shows favorable finding that Cells performance was increased by using the adaptive scheduling of task-level and loop-level parallelism and scheduling loop-level parallelism outperforms other notable processor, thus signifying that Cell's performance will scale as its programmer adapt to better schemes and techniques in programming it.

MOTION JPEG SERVER:

Muta et al. [3] take a similar approach to Blagojevic et al. [2], in deciding to take a standard application and implement it on the Cell architecture. Although the domain of the application is different (entertainment v. scientific communities), they both share a common thread in terms of the high amounts of parallelism they can exploit. Muta et al. note the need for a real-time high quality Motion JPEG encoding system, and that today's computers achieve about 1 frame every 2 seconds, about half of real-time speed.

The algorithm for encoding JPEGs under the JPEG 2000 ISO Standard is a four-part process. The paper is light on the details of each specific step, but does provide enough details for the causal reader to follow. There is some ambiguity in the paper,

however, as the first step, “Preprocessing”, is said only to be needed for lossless conversion, and that the third step, “Quantization”, is said only to be needed for lossy conversion. Therefore, there are always only three steps needed, in contrast with the paper’s claim that four steps are needed.

They begin by giving the obligatory explanation of the Cell architecture, and by implementing a simple version of the encoder that has no parallelism and only runs on the Power Processing Element (PPE). It performs very well, considering that compared to two open-source JPEG encoders, JasPer and OpenJPEG, theirs performs the best. This seems to be because they optimize their version for the PPE and make use of the PPE’s fast vector libraries, whereas the open-source solutions do not and rely on the compiler to do so. They give evidence to support this, showing later in the paper that JasPer and OpenJPEG on the Intel architecture greatly outperforms their implementation.

Muta et al. see that the last two steps of JPEG encoding, the wavelet and EBCOT steps, take the majority of the running time, and aim to offload this work to the SPEs. They begin with the wavelet step, and divide their JPEG into “pseudo-tiles” (we’ll call them “blocks” since we feel it is more descriptive) to process. In order to get the best performance, they make many design decisions that are completely dependent on this version of the Cell architecture. For example, they make the size of each block 128 square pixels, in order to maximize DMA transfer speed, which works best on multiples of 128 bytes. Furthermore, since they need to double-buffer the data and the SPE’s local store is 256 KB for data and instructions, they cannot increase the size of the blocks further than 128 square pixels. At first glance, this does not appear to depend on the Cell architecture too much. However, from our experience on the Cell, we’ve seen that this

data needs to be aligned before it can be used, that memory addresses to each block must be maintained by the PPE, and that the pertinent addresses must also be maintained by each SPE for data transfer. This shows that something as simple as passing around blocks requires much more knowledge and dependency on the Cell architecture than [3] lets on. We shall explore the true depth of this argument in the next section of this paper. Muta et al. hint at it by discussing how the SPE's registers are 16 bytes and that DMA transfers must be aligned manually to a multiple of 16 bytes, but do not go into it any further.

The wavelet step essentially performs a stencil operation, updating the data in each cell based on its neighbors. Another optimization done at this step is loop unrolling, a common technique used by the compiler. They use it to help the compiler optimize data fetches, but besides from the code ending up less readable (common to loop unrolling), the code is now dependent on the specific version of the compiler and architecture to achieve a consistent speedup. If future versions of the compiler perform loop unrolling on their own or other optimizations, this code will need to be refactored, and since it is less readable because of the loop unrolling, it will be more difficult to do so.

Thankfully, their implementation of the next step, the EBCOT step, is much less dependent on the compiler and architectures used. They do rely on 32 square pixel blocks for presumably the same architecture reasons previously discussed, but besides from that, the implementation appears to be very flexible. In this step, each SPE performs bit modeling and arithmetic coding, and their use of parallelism keeps each SPE busy (>90% utilization).

Muta et al. have an IBM Cell Blade Server containing two Cell processors, and decide to use each Cell chip as a separate resource. The main argument for this seems to

be that communication across Cell chips is much slower than staying on-chip, which is a valid claim. They also encounter a synchronization issue when using many SPEs to perform the wavelet transform, and thus decide to use one SPE for the wavelet transform and the remaining 7 for the EBCOT step. Resolving this synchronization issue may improve performance further, and resolving this would be another possible extension to this paper.

Three versions of the Cell Motion JPEG 2000 Encoder are then compared: the original PPE-only version, a version uses one PPE to direct the sixteen SPEs on both Cell chips, and a final version that two PPEs, each directing their eight SPEs. They also compare the number of frames per second that their encoding cluster (since it is more than one Cell it may be fair to call it this) can encode given 24 frames and 240 frames. Their results are unfortunately, completely intuitive. The version that only uses the PPE is by far the slowest of the three, and that the version that uses one PPE to coordinate 16 SPEs leaves too much coordination work to the PPE for it to be truly optimized. Finally, adding more blades results in a linear speedup in the number of frames that can be encoded per second. Since the application is optimized for one Cell processor, adding more independent threads (here blades), this result is unsurprising.

Muta et al. meet their goals in producing a real-time Motion JPEG 2000 encoder, although they need two Cell processors to do so. Although this is one blade, with just one Cell processor and linear slowdown, which we saw earlier in the form of speedup, we should expect about one frame every two seconds. This is exactly the same speed as the original processor, so it is unclear if there is actually any gain by using the Cell processor. I believe this paper is a good model to explain how to parallelize JPEG encoding, but it is

not convincing that much is gained by using Cell. The main gain is the linear speedup based on the number of Cell processors, and although this is notable, Muta et al. spend little time discussing it.

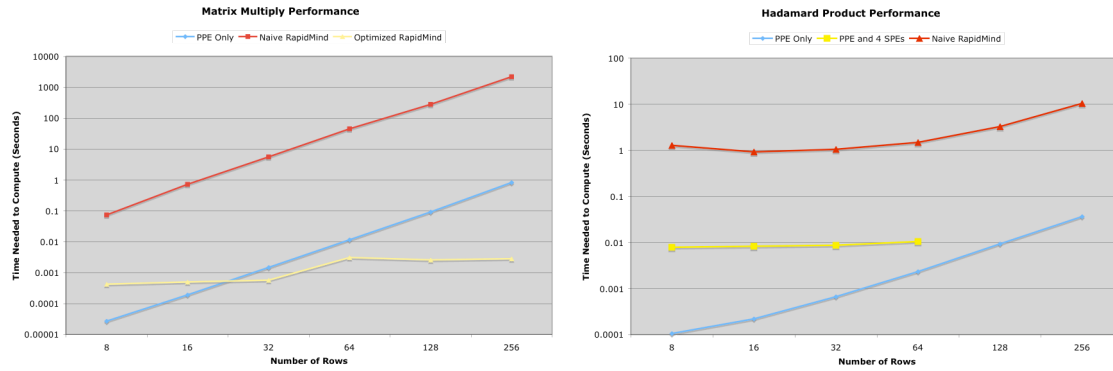
OUR CONTRIBUTIONS:

We believe the unique architecture of the Cell processor makes it worth studying firsthand. Whether it becomes accepted or not will ultimately depend on how easy it is for the average programmer to use it well. An important part of this is how much work the compiler can optimize for the programmer and how much work the programmer must optimize on their own. Furthermore, we see how the Cell performs on a given task using different compilers and how our code stacks up against optimized code.

For the purposes of this research we have obtained a Sony PlayStation 3 and installed Yellow Dog Linux 5 (YDL). Sony has supported YDL and made it the primary choice of Linux for the PlayStation 3, and IBM gives a thorough walkthrough detailing how to install it on the PlayStation 3. At some point, IBM Blade Servers seem to have become the preferred Cell development device, however, thus IBM transitioned to using and supporting Fedora Core Linux (as YDL is supported by Terra Soft). New versions of the Cell SDK are available for free for Fedora Core, but come at a heavy cost to Yellow Dog Linux users. Furthermore, the free version for Fedora Core does not install to our YDL system. There is ample documentation for installing Fedora Core on IBM Blade Servers, but none for installing it on a PlayStation 3. Doing so will be our next line of work. We attempted to install it on a number of other machines to cross-compile code for Cell, but the installer strictly requires Fedora Core Linux. Oddly enough, it could not be

installed on any of our Linux machines, which were running Gentoo Linux and Debian Linux. Since this SDK also contains the enhanced IBM Cross-compiler for C (IBM XL C), we were not able to use it either. Therefore, our research has been done using the slightly older Cell SDK 2.0 and the standard GNU C Compilers for the PPE and SPE (gcc and spu-gcc, respectively).

For the purposes of our comparisons, we use four different compilers / environments. The first uses the standard GNU C Compiler to produce code solely for the PPE. Since SPEs are not autonomous, the second uses the PPE to direct the SPEs. Then, we re-implement the algorithm using the RapidMind framework. It allows the programmer to take their existing C/C++ code and use macros and keywords to specify which parts of their program can be performed in parallel. Versions of this compiler produce optimized code for the Cell architecture and the GPU architecture, and while normally a commercial product, are offered free for academic and research purposes. We implement two simple algorithms for use on the Cell architecture: matrix multiplication and the Hadamard product. Matrix multiplication is a familiar algorithm, and the Hadamard product takes in two matrices and returns a new matrix with each element being the product of the two elements in that spot in the original matrices. Both of these algorithms have high amounts of parallelism and should see a favorable speedup on the Cell architecture. Finally, we compare the performance of the code we've written versus optimized code provided by RapidMind. Since this optimized code was only available for matrix multiplication, we have no optimized code for the Hadamard product and it is excluded from our graph. Our results are as follows:



There are a few things to note here. Both graphs use a logarithmic scale on the y-axis. For the matrix multiply example, we were unable to send more than 16 KB to the SPEs, and thus could not use the second compiler / environment, PPE directing SPEs. We run into the same issue with the Hadamard product implementation, so we only graph the sizes below 16 KB. Furthermore, RapidMind does not produce an optimized Hadamard product algorithm, so we only compare our implementations. Finally, we are only timing how long it takes to perform the computation. This is to match the RapidMind optimized code, and their recommendation only to time the computation. We can see from the first graph that both of our implementations scale in size exponentially, proving that we are far from expert programmers. The optimized RapidMind code scales much better than our code does, and simply using RapidMind without truly understanding how to use it well has disastrous results. In our case, our naïve RapidMind implementation is worse than our PPE-only code by three orders of magnitude (1000x worse)! Furthermore, this difference in performance scales up exactly in the same fashion as the PPE-only code.

The Hadamard product implementations fare about the same. The PPE-only code and naïve RapidMind code scale exponentially (especially towards the bigger matrices), but the PPE directing the SPEs performs moderately well. We elected to only use four SPEs instead of the six available (not eight, as one is turned off to meet die yield goals

easier and one is used by the OS and hypervisor) because dividing up a square matrix works much easier when dividing it into four pieces instead of six. Regardless, its performance is what we would expect: it performs poorly when there is not enough work to ship to the SPEs, and performs well when there is. Once we find an easier construct to send more than 16 KB to each SPE, we intend to revisit this issue and complete this graph.

Developing in Cell is definitely a different experience compared to more standard architectures and languages. The architecture exposes itself to the programmer, allowing them to do tasks normally reserved (and thus not done) by the compiler. This is both Cell's greatest potential for success, and (currently) it's greatest downfall. It allows expert programmers to get great performance out of it, but at this time, forces you to be an expert programmer to get anything done on it. Our future work on Cell will surely involve making it easier for the average programmer to write working code that takes advantage of the SPEs, even if it is not to the fullest extent possible. Even something as simple as sending more than 16 KB of data to an SPE is obscenely difficult, but programmers that can do this can be considered experts. In contrast, the CUDA language exposes many low-level details but does not force the user to schedule DMA transfers, partition their data into 16 KB chunks, or align their data in memory as a multiple of 16. Those three issues are particularly difficult on Cell, and the amount of knowledge available to do these is particularly sparse.

The IBM tutorials give the most basic of introductions and refer the user to the Cell B.E. Programming Handbook [4], a nearly nine hundred page manual that gives an excellent walkthrough on how to program the Cell architecture in assembly code. The

much shorter “C/C++ Language Extensions for Cell Broadband Architecture” [5] focuses mainly on manipulating vectors, and is completely lacking in terms of aligning and sending data, the hardest tasks on Cell. Help can be found on the IBM Cell Forum, but the users and administrators assume you to be an expert Cell developer, so this is also a dead-end.

RapidMind is designed to provide an alternative to the issues plaguing development on Cell. As opposed to having separate files containing code for the PPE and SPE, RapidMind allows programmers to have one file with annotations on parallel snippets of code. Like Cell, the programmer will have to restructure their program with the parallel code in a separate function, and RapidMind uses special data types. Even primitive data types in parallel sections need to be replaced by their parallel counterparts. The biggest drawback from using RapidMind is the sheer difficulty in debugging compiler warnings. For example:



```
cgb@cell:~/hadamardAnalysis — ssh — 80x25
withRapidMind.cpp: In function 'int main(int, char**)':
withRapidMind.cpp:40: error: no match for 'operator[]' in 'matrixA
[rapidmind::Value<du, int, rapidmind::General, false>(row, column)]'
/usr/include/rapidmind/platform/ArrayImpl.hpp:173: note: candidates are: rapidmi
nd::ArrayElement<T, du, int> rapidmind::Array<D, T>::operator[](int) const [with
unsigned int D = du, T = rapidmind::Value<du, int, rapidmind::General, false>]
withRapidMind.cpp:46: error: no match for 'operator[]' in 'matrixB
[rapidmind::Value<du, int, rapidmind::General, false>(row, column)]'
/usr/include/rapidmind/platform/ArrayImpl.hpp:173: note: candidates are: rapidmi
nd::ArrayElement<T, du, int> rapidmind::Array<D, T>::operator[](int) const [with
unsigned int D = du, T = rapidmind::Value<du, int, rapidmind::General, false>]
/usr/include/rapidmind/platform/ArrayImpl.hpp: In constructor
'rapidmind::Array<D, T>::Array(unsigned int, unsigned int) [with unsigned int D
= du, T = rapidmind::Value<du, int, rapidmind::General, false>]':
withRapidMind.cpp:31: instantiated from here
/usr/include/rapidmind/platform/ArrayImpl.hpp:74: error: no matching function
for call to 'rapidmind::CompileTimeChecker<false>::CompileTimeChecker
(rapidmind::Array<D, T>::Array(unsigned int, unsigned int) [with unsigned int D
= du, T = rapidmind::Value<du, int, rapidmind::General,
false>]::ERROR_Array_Dimension_Does_Not_Match_Constructor_Parameters&)'
/usr/include/rapidmind/platform/Utility.hpp:30: note: candidates are: rapidmind
::CompileTimeChecker<false>::CompileTimeChecker()
/usr/include/rapidmind/platform/Utility.hpp:30: note: candidates are: rapidmind
::CompileTimeChecker<false>::CompileTimeChecker(const rapidmind::CompileTimeCheck
er<false>&)
```

Passing a one-dimensional array to a function that was expecting a two-dimensional array produced this error. The equivalent compiler error message using the standard Cell C compiler would have been one line long and clear about what the problem was. This message gives the programmer absolutely no information to let them

know that a two-dimensional array was needed, and is completely inexcusable, as this type of message constantly comes up for all compile time issues.

CONCLUSION:

The Cell architecture is truly an interesting leap forward in the arena of parallel computing, but key design and implementation choices in the various compilers available make it unrealistically difficult to program on. We have showed various environments that enable the programmer to develop on Cell and the downfalls surrounding each of them. We intend to extend this work and provide solutions enabling the average programmer to work effectively on the Cell architecture. Furthermore, we wish to use this work in a way that can be extended to other parallel architectures, forwarding the entire arena of parallel computing with us.

ACKNOWLEDGEMENTS:

We wish to thank Fred Chong for graciously supplying us with a PlayStation 3 with which to put Linux on and conduct the experiments seen in this paper, and for allowing us to use his Architecture Lab to house it in.

REFERENCES:

- [1] Bellens et al., “CellSs: A Programming Model for the Cell Architecture”, *IEEE Press*, 2006
- [2] Blagojevic et al, “Dynamic Multigrain Parallelization on the Cell Broadband Engine”, *ACM Press*, 2007

[3] Muta et al., “Multilevel Parallelization on the Cell/B.E. for a Motion JPEG 2000 Encoding Server”, *ACM Press*, 2007

[4] IBM, “Cell Broadband Engine Programming Handbook” [Online], < http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine>, 2007

[5] IBM, “C/C++ Language Extensions for Cell Broadband Engine Architecture” [Online], < http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine>, 2007

[6] Jeff Atwood, “Performance Considered Harmful” [Online], < <http://www.codinghorror.com/blog/archives/000082.html>>, 2004.