

System Support for Multiple Heterogeneous Cores

CS5204 Course Project: Fall 2007

Benjamin Rose
Department of Computer Science
Virginia Tech
bar234@vt.edu

ABSTRACT

Heterogeneous processing cores in a single system are becoming more popular as their improved performance and reduced power consumption gain notice. Extra accelerator cores are added to existing multi-purpose CPUs to offload computationally heavy work. These accelerator cores are typically designed for this kind of work so the result is an increase in performance with less work on the main CPU.

Software support for these architectures typically demands a lot more from a programmer and the operating system. Traditional types of parallelization have to be re-evaluated for use with these accelerators. A few tools are available to help the programmer with this task, however they are very new and require more work to fully utilize these accelerators. The OS also has to adjust its scheduling algorithms in order to accommodate for various performance capabilities of each core. The OS also has to worry about memory management since caches are bound to vary between these heterogeneous cores. Many human and non-human factors have to be taken into consideration when attempting to fully utilize a heterogeneous core system.

1. INTRODUCTION

The idea of using different types of processing cores in a single system is something that should be considered now that the industry is becoming more comfortable with homogeneous multicore processors. The idea is a very basic one that has apparent advantages. For example, today's off the shelf systems all include Graphic Processing Units to help render the graphics that appear on our screens so the CPU doesn't have to. Why shouldn't we add other processing units to help with other specialized tasks?

Embedded systems already use heterogeneous cores on a single system[7]. Re-programmable FPGAs are used to supplement the main processing unit when a computation has to be performing many times which would normally bog down the main processor. This computation can be programmed

into the FPGA and whenever that computation is needed, the data is offloaded to the FPGA. This frees the processor to continue doing other tasks or scale down its frequency. When the FPGA is idle, we even have the option of completely shutting it off to save power, a very essential part of some embedded systems.

Using heterogeneous cores on a single system is nothing new, but using them in very fast and very powerful systems to reduce cost and improve performance is. One major obstacle is the adjustments needed to be made by the programmer. Programs have to be written so different behaving parts are able to be executed on different cores. Sometimes this means compiling different binaries for different parts of the code, like for the Cell Processor, or just adding extensions and library calls to ensure one area of code is executed on a GPU instead of the CPU. Programmers have to be aware of available accelerators when writing programs to fully utilize the extra cores.

The operating system also has to make adjustments for heterogeneous cores. Schedulers can't assume each processor now computes at exactly the same speed and with the same ISA. Each core has to be sampled with various pieces of code in order to determine which processes perform best on which core. The operating system also has to worry about memory management for these different cores. Typically heterogeneous cores also means heterogeneous caches. NUMA has to be considered and adjustments to the operating system structure should be made to work best with heterogeneous cores.

This paper will begin by introducing some hardware architectures used in heterogeneous chip systems in Section 2. Section 3 will be devoted to the difficulties of programming on heterogeneous cores and how a programmer can implement various levels of parallelism on such a platform. Section 4 will discuss language and compiler support to help the programmer utilize the different kinds of cores. Section 5 discusses what the OS must provide in order to allow applications to use different cores. Section 6 focus on a case study of the Cell BE and its benefits and drawbacks. Lastly Section 7 will make conclusions and state insights based on content discussed in the paper.

2. HARDWARE

There are a few different ways of arranging multiple heterogeneous cores in a single system. This section will take a

look at some of these architectures and hardware obstacles that have appeared.

2.1 Architecture

Two main hardware architectures of heterogeneous cores examined are placing multiple heterogeneous cores on a single chip, and utilizing a GPU as an accelerator for the CPU. The direction of chip manufactures toward multiple core chips inspires us to investigate the possibility and results of using various kinds of cores on a chip. While another approach considers using the GPU as an accelerator, since the hardware is already present, and the newest GPUs are proving to provide exceptional computation power.

One of the most desired architectures is placing multiple heterogeneous cores on a single chip. This not only improves communication between the chips, but makes multiprocessor systems easier to manufacture since we can use multiple cores in one socket.

Figure 1 illustrates the layout of the entire Cell Processor and shows details about each Synergistic Processing Element (SPE). The SPEs contain all the necessary components of a processing core (Local Store Cache, Floating Point Unit, Registers, Instruction Logic, etc.) but are much smaller than the Power Processing Element (PPE), which is much closer to the traditional processor cores in today's computers. In order to reduce the physical size of the core, the local cache is small and contains only the most basic instructions.

When using different cores on the same chip, a question arises regarding the instruction sets of all cores. Should all of the cores use the same ISA but differ in size and speed or should we use cores that have moderately/completely different ISAs? The advantage to using the same ISA is when efficient power usage is critical, as we can switch off the power hungry cores and run the same programs on smaller, lower-power cores[6]. Also, the programmer doesn't have to worry about different compilers and multiple binaries for a single program. However using different ISAs allows us to run various kinds of code on optimized cores. It also helps hardware designers to create accelerator cores that contain only the bare minimum they need to perform computations quickly. If they had to worry about maintaining ISA between all cores, they might have to add extra features and instructions to these cores that really aren't needed, unnecessarily increasing core size and power requirements. Architectures with the same ISA between cores are easier to program and can reduce power consumption vs. a homogeneous model, however multiple ISAs might be more popular if hardware designers don't have to build in backwards compatibility with existing ISAs but can instead introduce their own.

Another approach to multiple heterogeneous cores that has been investigated is utilizing the GPU as an accelerator core. Since the GPU is designed for a large number of complex graphical calculations, it has many different processors for vertexes and pixels. These processors result in a large performance capability for the GPU[10].

Using a GPU as an accelerator has not been exploited much because of the closed structured of the GPU. The only way

to access the computing power of most GPUs is by utilizing graphic APIs (such as OpenGL). Also, when accessing the GPU, data has to travel over an external bus from the CPU to the GPU, which can cause extra latency[10]. This has to be considered by the programmer so communication between the units is minimized.

3. PARALLELISM ON HETEROGENEOUS CORES

Programmers have to adjust their models of parallel programming to accommodate and fully utilize heterogeneous cores. When not using tools to help distribute the loads to various cores, or when unable to, the programmer has to manually control which cores run specific areas of code. It is also up to the programmer to ensure the code that runs on each specialized core is optimized to perform best on that core.

3.1 Task-Level Parallelism

A high-level programming model for heterogeneous cores involves off-loading whole functions onto accelerator cores, usually run as a separate thread from the main thread. These functions are the computational heavy parts of applications and typically take up the majority of execution time when on a single core. Since this computationally heavy function is executing on an accelerator, simply waiting for the accelerator to finish that function should yield speed up. This is one of the easiest, and highest levels, of parallelization.

When using this model with heterogeneous cores, the programmer has to be able to designate tasks to be run on cores that are best designed to handle these offloaded functions. For example, the rendering of graphic interfaces is typically offloaded to the GPU. The programmer might have to define a few different versions of the same function to run on various types of cores, to avoid overloading one or few cores of a specific type (Perhaps we can render graphics on a generic accelerator core if the GPU is busy). Although ideally only one function and core combination will perform the fastest, running slightly slower versions of the function on other cores allows more data to be processed in parallel, thus the program isn't waiting on the optimal function to run on the optimal core.

3.2 Loop-Level Parallelism

On a finer-grain level than Task-Level, Loop-Level can be implemented on top of Task-Level parallelism to utilize any idle accelerator cores. Reducing the number of iterations each accelerator has to execute in a large, many iteration loop, reduces the time needed to finish the computation. Distributing these iterations across any idle accelerators can help achieve this. For example, when using the Cell BE, once a task is assigned to run on a SPU that SPU can break the task up by dissecting the loops[4]. This allows for a work-sharing model between SPUs with a task and idle SPUs.

Implementing Loop-Level Parallelism faces the difficulties resulting from the fact not all cores are equal in a heterogeneous environment. The programmer has to consider how many loop iterations each core can process so all cores finish the whole loop at approximately the same time. If the

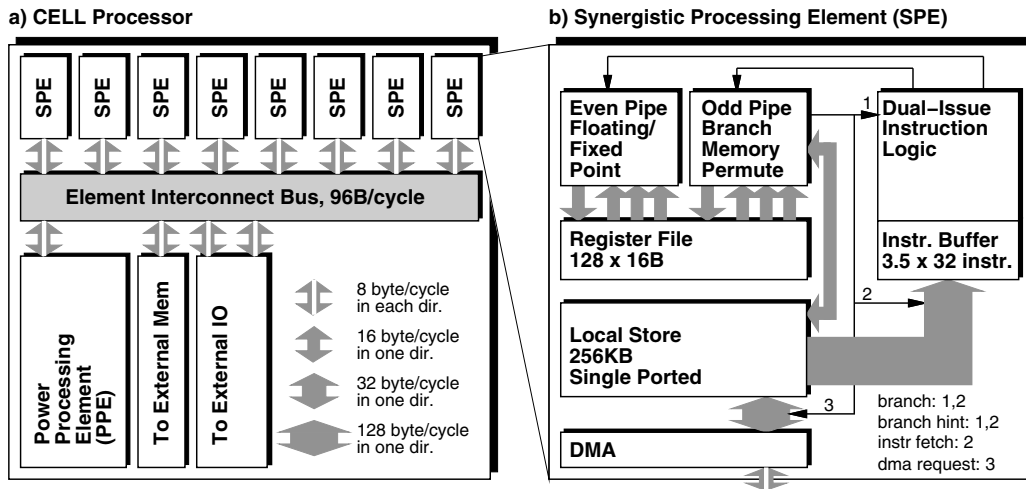


Figure 1: First Generation Cell Processor. Note that each SPE has its own processing elements[12].

programmer has specific knowledge of the cores the application will be running on, this isn't quite as difficult, it just requires some fine tuning of the code.

However if the programmer has no knowledge of the cores a loop may be spread out onto, a large amount of extra code might be required to eventually achieve equal processing time for all the cores. A sampling mechanism could be introduced to test how many loop iterations each core can perform in a set amount of time, and based on this sample the program can distribute the work accordingly. However this in itself introduces extra overhead as samples have to be determined and face communication delays as the samples are sent to the cores and results are returned. If implemented correctly, an application could see enough speed up using a sampling mechanism that the overhead can be considered tolerable.

Another idea is to allow for dynamic chunks of iterations be distributed to all cores. When a core finishes a chunk of iterations, it requests more work. This happens until all of the iterations are finished. This allows for fine tuned load balancing, by specifying the chunk size, however it also has overhead in the form of extra communication. The chunk size and number of work requests are inversely proportional.

The final decision of how to use loop-level parallelism has to be done by the programmer after considering which model will yield the best performance for the individual application.

3.3 SIMD Vectorization

On the lowest level, we have to make adjustments to operations in order to accommodate for Single Instruction Multiple Data (SIMD) architectures. These architectures are able to perform the same operation on multiple pieces of data simultaneously. Many kinds of accelerator cores and the vertex shading processors in most GPUs are SIMD architectures. To fully utilize this the programmer must try to fill up the data vector so the accelerator core processes as much data as possible per clock cycle[3]. This is where a

lot of speed up will be noticed. Used in an efficient manner might require the programmer to handle a few loop iterations at once.

When vectorizing code that is to be run on SIMD accelerator cores, it is important to know the abilities of that specific core. The biggest concern is the size of the data vector that a single instruction can be performed on. For example, the SPU in the Cell has a 128 bit data vector that, when fully filled, can dramatically improve computation time. If, in a hypothetical sense, we have another core next to the SPU that has a 256 bit data vector but everything else is the same as a regular SPU, computation time could be cut in half if the application accommodates for this larger data vector size.

Knowledge of the various cores by the programmer is essential for maximizing efficiency when an application is exploiting SIMD Vectorization. As suggested with Loop-Level Parallelism, it might be possible to implement a sampling mechanism, however this depends on the architecture of the core. If we are able to use any size of vector and have the hardware split up the vector into chunks that utilizing the maximum data size, then a sampling mechanism would work out well. Or the abstracts provided to access the SIMD capabilities of that core might limit the data that can be operated on at a time. It highly depends on the abstracts provided that determine what we can do with the individual core's SIMD capabilities.

Another possibility is if the programmer knows a large range of cores the application could be run on, the application could adjust the vector size according to parameters in a table the programmer has implemented. If the process can tell what core it is running on when it begins, it is possible to look up these parameters in a table and use these.

3.4 Pipelining

Pipelining is a method of processing data in a streaming fashion. Each accelerator core performs certain operations on global streams of data. Streams of data are processed by

each core, usually in a specific order. When core 0 finishes processing the first chunk of data and passes it to core 1, it can begin work on the second chunk of data. This process continues: when the cores 0 and 1 are done processing the second and first chunks, respectively, the first chunk is handed to core 2, the second chunk is handed to core 1, and core 0 begins work on the third chunk. This process continues until there are no more streams of data to process.

Each core should be implementing SIMD vectorization if it can perform SIMD operations for maximum speed up. Also, communication buffers need to be of optimal length so they do not take away too much memory from the computations, but are big enough to hold incoming chunks the previous core finishes.

The difficulties of implementing pipelining on a heterogeneous core environment are significant. Due to the way pipelining is organized, all data streams are computationally bound by the slowest process. A few ways to get around this is to give the process less to do or have a couple cores perform the same process, however when the environment is heterogeneous and the programmer is not targeting a specific platform, this becomes much more challenging.

As mentioned in the previous difficulties, a sampling mechanism could be used to determine how the work should be distributed. However, because of the nature of pipelining, the sampling structure would be much more complex than sampling for Loop-Level Parallelism or SIMD Vectorization. A collection of tasks to be performed on the data stream could be executed on each core, thus creating a ranking of which cores performed which tasks the best. This sample could then be used to assign multiple cores to work on a single task, if there are fewer tasks than cores, but care has to be taken that data flow in and out of each task remains approximately constant. If a task is performing slower than the others, data flow has to stop in the earlier tasks until the slower task finishes. These fine tuning adjustments could mean a lot of extra implementation code.

While many of the solutions suggested to adapt various parallelism models to heterogeneous cores include approaches that require the application to sample each core, this might not be possible or be infeasible for some kinds of applications. Therefore the programmer might find extra tools useful that can automatically allocate the processing resources and utilize them efficiently.

4. LANGUAGE, COMPILER, AND RUNTIME SUPPORT

This section discusses various kinds of libraries and compiler support for utilizing multiple heterogeneous cores in a single program. These tools are designed to aide the programmer during application development. We examine the most popular methods of parallelism in C# and C/C++ with OpenMP and suggest tools that help exploit heterogeneous cores.

4.1 Accelerator

When programming with C#, task level parallelism is of the utmost importance. Putting the GUI, I/O, and var-

ious other tasks on their own threads is important for a responsive application. C# provide easy to use threading mechanisms to help the programmer separate the tasks in the application. Modifying this to utilize a heterogeneous system is not a very far stretch.

Accelerator was developed by a few members of Microsoft's research group to provide an API in C# for offloading tasks to the GPU[10]. Since the GPU has many different pixel shader processors that perform operations on data in the GPU's texture memory, the Accelerator API adjusts ordinary data so that it appears as a texture in the GPU's memory. The API primarily provides new data types, that when used for computations are compiled into textures as the program runs in order to be executed on the GPU.

Accelerator is still a very recent tool and as such it has many optimizations that can be made to improve performance.

4.2 EXOCHI

One of the most popular ways to parallelize large chunks of computations C/C++ code is to implement loop-level parallelism. This is most commonly done by using OpenMP, as it provides the programmer with compiler directives to mark sections of code that should be executed as multiple parallel threads[11]. While this works out very well for homogeneous cores, since the work is typically partitioned into equal chunks, we need a few modifications to fully utilize heterogeneous cores.

EXOCHI provides two significant tools for programmers to help program for heterogeneous accelerators[11]. The first is the Exoskeleton Sequencer (EXO) which wraps heterogeneous cores so they appear as resources in the actual program code. The second is the C for Heterogeneous Integration (CHI) which uses the accelerator resources provided by the EXO with abstracts to designate which cores specific sections of code should run on.

EXOCHI helps programmers utilize multiple heterogeneous cores in the same way OpenMP helps programmers use homogeneous cores. However the usability comes at the price of not knowing much about the underlying hardware. Communication heavy operations has shown to perform poorly on certain performance tests using EXOCHI, and the programmer might be better off using tools specific to that accelerator core to manually optimize communication.

4.3 Improvements

Both Accelerator and EXOCHI are relatively young and their use is only experimental. Further work needs to be done on both in order to seriously consider them as possible candidates for production application development. Accelerator provides a very easy to use interface for the programmer, however its execution could be a lot better. Future versions might deliver more promise but until then programmers are stuck either writing vertex shader code by hand to handle their computations or hope the less than optimal performance of Accelerator is still greater than not offloading the work all together.

EXOCHI is also in a similar situation. It provides a very easy way to utilize, in theory, any sort of accelerator core.

The ideas are very good and the current results are promising, however there is still work done, especially considering its age. Communication heavy work still performs quite poorly with EXOCHI and further work needs to be done to improve this. Perhaps building in some mechanisms to allow for double buffering, so data can be requested far enough in advance that its latency is not noticed. This is no easy task, however if accomplished EXOCHI would become a very viable candidate for heterogeneous core development.

With or without extra tools, libraries and runtimes, the performance and capability of a program largely relies on the platform upon which it runs.

5. OS SUPPORT

The Operating System has to worry about a few extra parameters when the system consists of multiple heterogeneous cores.

5.1 Loading Programs

The Cell BE is able to run the Linux operating system since the PPE is based on a PowerPC architecture. The tools and libraries provided by IBM help the programmer manage the code that is run on the SPEs. However the Linux kernel that is run by the Cell is modified from the vanilla kernel so the Operating System has minimal knowledge of the SPEs (mostly just where they are and how to send code to them), only enough so the IBM provided libraries can do the rest. Current work is being done to represent the SPEs as a virtual file system, however this is still a work in progress[2].

While this is one approach, another could be to compile and run the bulk of the Operating System on a single ISA but build in support for a selection of other ISAs. If the Operating System is able to read a program, decipher what architecture it should be run under, and place it on a run queue for cores of a matching architecture, it would allow for a mix of architectures among running processes.

5.2 Memory Management

If the Operating System supports different architectures, it would have to be mindful of memory accessibility. In the case of a General Purpose GPU, data has to be transferred across the system to the GPU's memory before any operations can be performed on it. The operating system has to know that any memory that GPU allocates should be using a page table for that GPU's physical memory, and not the system memory. Paging across a long distance multi-purpose bus might experience more latency than the main CPU paging with main memory, so the programmer would have to keep that in mind when developing the application.

The SPUs on the Cell also have special restrictions when accessing memory. Each SPU has a local storage of only 256k, this can fill up quickly. The SPUs have access to main memory via a DMA interface. The Operating System could implement a page table for each SPU, where "paged out" pages would go to main memory instead of disk. Then the main page table for the main memory can handle paging to disk. This can also be extremely taxing if the SPU needs a page that is on disk, since it would have to travel through two page in operations, one to bring it into main memory and another to bring it into the local storage.

This two tiered paging approach also could work for any kind of accelerator core. When using one giant address space for virtual memory, at worst two page faults can trigger. One to load the page into main memory, and another to load the page from main memory into the core's local store.

If an efficient paging and virtual memory system needs to be devised so the programmer doesn't have to worry about custom memory management, that can result in another hurdle cleared for heterogeneous core programming.

5.3 Scheduling Tasks

This section focuses primarily on scheduling between heterogeneous cores of the same ISA. Scheduling for different ISAs wouldn't be very difficult as each ISA could have its own run queue, which then distributes tasks to each core of that specific ISA (if there are more than one). Sampling mechanisms explained in this section can also be applied when there is a mix of ISAs, but a large number of cores of the same ISA in that mix.

5.3.1 Concept

Creating schedulers that are aware of heterogeneous cores is a task that has been investigated with a few proposed solutions. The general idea is to use a sampling technique of running each task on each core for a brief period of time (relative to the time to execute the whole task) and using these results, tasks are scheduled on the cores where they perform the best. These schedulers assume all tasks and cores are of the same ISAs.

Nollet, Coene, Verkest, Vernalde and Lauwereins introduced a paper describing an OS they developed that was able work with a heterogeneous reconfigurable system[8]. The biggest contribution of the paper is a scheduler that uses a two-level scheduling technique to assign tasks to heterogeneous processors. The top level decides which task should run on which processor. The lower level involves running local schedulers on each processor, so that specific processor is able to fairly run all tasks assigned to it. Each scheduler (top and lower) assigns each task a timeslice before that scheduler re-schedules that task. The top level scheduler has the ability to move tasks between processors, however this can only occur at specific programmer-defined checkpoints in the task. It is up to the programmer to ensure context information that is stored and transferred with the task at a checkpoint is minimal.

5.3.2 Performance

A simulated heterogeneous configuration of Alpha cores was constructed by Kumar, Tullsen, Ranganathan, Jouppi, and Farkas in order to test a scheduler for a heterogeneous environment and provide benchmarks against a few homogeneous systems[7]. Their scheduling mechanism is similar to the one in Concept. Sampling each task on each core is done to determine the best pairing and scheduling is performed based on these results. Various benchmarks compare a static scheduling mechanism and the sampling scheduling mechanism versus standard scheduling on a homogeneous environment.

Various heterogeneous scheduling mechanisms (using three EV6 and five EV5 Alpha processors), plus a homogeneous

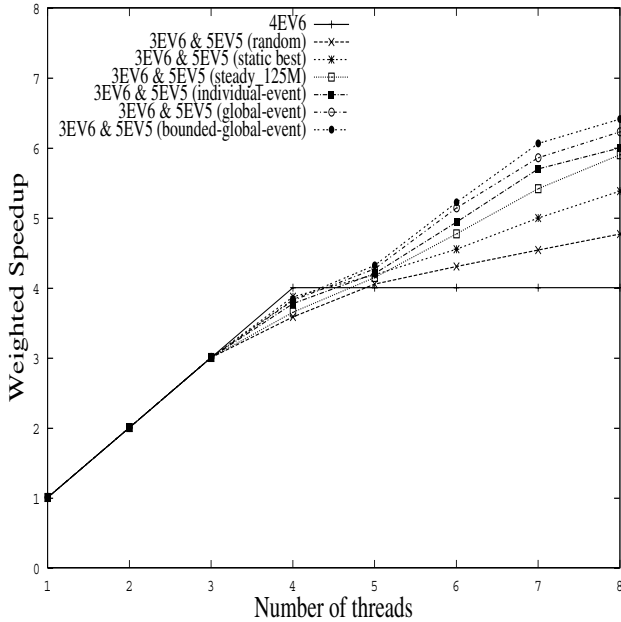


Figure 2: Homogeneous vs. Heterogeneous speed up using various trigger conditions for resampling[7].

configuration (four EV6 Alpha processors), are tested to determine the best configuration. *Random* scheduling places a task on a random core with no sampling or monitoring. *Static best* uses a static heuristic where the more powerful cores are used first. *Steady_125M* represents sampling at every 125 million cycles unconditionally. *Individual-event* represents resampling whenever that thread’s steady-phase IPC suddenly changes by more than 50%. *Global-event* is similar, except it looks at the change in everyone’s IPC and if this change is greater than 100%, a resample is triggered. Lastly, *bounded-global-event* is very similar to global-event, however it also considers the cycles that have passed since the last resample. If an event is triggered but it hasn’t been more than 50 million cycles since the last resample, the event is ignored. Also, if more than 300 million cycles pass without a resample, one is manually triggered.

Figure 2 shows the results of running these different triggering mechanisms. We see bounded-global-event is the best scheduling mechanism performance-wise. This is not surprising since this mechanism is incorporating the best trigger-based resampling mechanism with a cycle counting resample mechanism.

Scheduling in a heterogeneous environment using relatively simple mechanisms can yield great performance increases. The performance tests in this section show the performance gains of applying simple scheduling rules over a homogeneous environment. Adding a couple rules help determine when to resample gets even more performance benefits.

These scheduling mechanisms can be easily applied to multiple cores with different ISAs. As stated before, using two-level run queues allows for a top-level run queue for each ISA. These top level run queues then divide up the work onto core-specific run queues that match the top level run

queue’s ISA. With support for individual accelerator processes in Linux, applying these scheduling rules can provide a very heterogeneous-core friendly environment.

6. CASE STUDY - THE CELL BE

The Cell Broadband Engine, developed by IBM, Toshiba, and Sony, consists of a main PowerPC-based processor (PPU) and 8 accelerator cores (SPUs) of a different ISA[1].

6.1 Hardware

The Cell was constructed so it could run as a single two-way PPC core with optional accelerator cores (SPUs). The main PPC core uses a common 64-bit PowerPC architecture. The SPUs use a different, specialized architecture since the SPUs are physically much smaller and provide fewer specialized instructions and features than the PPU. The SPUs also operate in a SIMD fashion in order to provide faster computational performance. All of the cores are on a very fast Element Interconnect Bus (EIB) which provides communications between cores and to external components, such as I/O, ethernet, and main memory. Each core has a DMA element that moves data between cores and main memory.

One obstacle that appeared while designing the Cell was heat dissipation. The generation of heat from many small hot spots instead of a few centralized hot spots caused trouble for the spreading of the heat across the silicon substrate[9]. It took extra analysis and design adjustments to ensure maximum and average heat didn’t exceed thresholds. Once the design was adjusted, conventional thermal sensors were placed throughout the chip to control active cooling.

6.2 Application Development

When developing applications for the Cell BE, various approaches are taken. Typically when a well known application is ported to the Cell, that program is initially ran entirely on the PPU with profiling code added. This profiling code describes what pieces of code are taking the most amount of time to execute. The functions that take the most amount of time to execute are targeted for offloading on to the SPUs.

This task-level parallelism allows the PPU to worry about other housekeeping components of the application while the SPU executes the heavy lifting. Once the code is off loaded to a few SPUs, the code can be adjusted to utilize loop-level parallelism or data decomposition to utilize any remaining idle SPUs. DMA transfers between cores, not just a core to main memory, is a popular mechanism to achieve synchronization. The local storage of each SPU is given an address range lower than main memory’s address range which can be accessed via regular DMA calls.

Each SPU core is of a SIMD architecture, which requires the programmer to manually vectorize the data being computed in order to achieve maximum performance. A group of researchers at IBM have developed XL, a compiler that performs extra optimizations, such as automatically vectorizing computations, automatically when the code is compiled[5]. Also discussed are compiler techniques to automatically spread a single source program across all SPUs, but these techniques are not yet integrated into the compiler. Many of the programming models discussed in Section 3 are helpful in utilizing all the SPUs in the Cell BE.

6.3 Operating System Support

Since the Cell is still a relatively young technology, OS support is in heavy development current. Linux is capable of running on the Cell Processor since the PPU is just a PowerPC architecture chip. However little development has been accomplished to support the SPUs.

The most promising development for SPU support is SPUPS[2]. This provides a way of accesses each SPU via a mechanism similar to the procfs. Each SPU is assigned a directory, and inside of the directory various virtual files are used to access different components of the SPU. A mem file provides access to the local storage of the SPU and can be written into using regular file operations. Another file, run, provides a way to start and stop execution and either tell the SPU where to begin executing or return the place last executed when stopped. The last few files provide access to the SPU's in and out mailboxes. Eventually these can be used with the scheduler to help schedule many SPU-only processes inside of Linux.

6.4 Performance

When fully optimized, a program can perform exceptionally well on the Cell BE. With a theoretical peak of 204.8Gflop/s on single-precision operations, this far outperforms most single chip processors. Running variety of matrix tests shows about a 8.4x speedup over Opteron and Itanium2 processors, and about 2.7x speedup over a Cray X1E[12].

Since the double precision operation on the SPE performs so poorly (mostly because of the lack of double point precision operations in video games, where the Cell BE is being marketed) a few changes were proposed to help increase performance[12]. These changes result in a DP operation dispatch every other cycle instead of every 7 cycles with just a slight increase in surface area and heat dissipation. When modeled, the DP performance went from 1.83 GFlops/sec to 6.4 GFlops/sec. While still no where near the performance of SP, 25.6 GFlops/sec, it is a welcome improvement with insignificant hardware changes.

7. CONCLUSIONS

The introduction of multiple heterogeneous core instead of homogeneous in a single environment caters to the different demands of applications. Allowing cores that specialize in a particular type of computation to run tasks that are heavy with these computations results in significant speedup and lighter load on other cores in the system.

The kind of heterogeneous environment varies. The key tradeoff observed is how easily a programmer can write for a particular environment efficiently (utilizing all possible resources) tends to have an inverse relationship with the performance abilities of the environment. The Cell Broadband Engine has a very specialized and powerful heterogeneous environment, but it requires its programmers to compile two different binaries for each type of processor. Single-ISA heterogeneous environments allow very little modifications to the code to use the environment to see performance increases, however these performance increases are not close to the performance increases of the Cell.

Various levels of parallelization have been modified to work

more optimally in heterogeneous environments. However each level of parallelization brings with it more difficulties the programmer must overcome. Sometimes a sampling mechanism is needed to determine the right amount of work per core to achieve approximately equal load balancing. However these sampling mechanisms are not possible in some applications and might be impractical in others (where time to sample might be comparable to time to execute the whole application).

Tools are available to help the programmer utilize heterogeneous cores. EXOCHI does for programming heterogeneous cores what OpenMP did for programming homogeneous cores. Accelerator is another good tool for utilizing heterogeneous cores in C#, which can be used to offload whole tasks. Rendering a C# GUI entirely on the GPU could prove to be very beneficial. However both of these tools are very young and still in heavy development. Once these tools reach their full potential, their utilization could become essential efficiently programming for a heterogeneous core system.

Operating system support has some good ideas proposed but needs more development. Load balancing using the sampling mechanism in the scheduler has shown performance increases, however this needs to be adapted for real systems and for multiple ISAs. Using a two tiered paging system can also help alleviate problems of transferring required data to each core but without fully consuming that core's local storage. These are ideas worth investigating as they could prove to be very beneficial to OS support for heterogeneous cores.

Hardware manufacturers seem to be testing the waters of creating heterogeneous multi-core chips as the popularity of homogeneous multi-core chips has been on the rise. It is up to the software programmers to adapt their programming styles to these different environments to fully utilize them, even though tools can help a programmer in this task, we are still a far way from any sort of tool that will make programming for a heterogeneous environment as easy as a homogeneous environment.

8. REFERENCES

- [1] Cell broadband engine architecture and its first implementation.
<http://www.ibm.com/developerworks/power/library/pa-cellperf/>.
- [2] Arnd Bergmann. Spufs: The cell synergistic processing unit as a virtual file system.
<http://www.ibm.com/developerworks/power/library/pa-cell/>.
- [3] Filip Blagojevic and Dimitris S. Nikolopoulos. Exploring programming models and optimizations for the cell broadband engine using raxml. *2006 Virginia Tech High End Computing Challenge*, 2006.
- [4] Filip Blagojevic, Dimitris S. Nikolopoulos, Alexandros Stamatakis, and Christos D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 90–100, 2007.
- [5] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden,

- Daniel A. Prener, Janic C. Shepherd, Byongro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for a cell processor. *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 161 – 172, September 2005.
- [6] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. *Microarchitecture*, pages 81–92, December 2003.
- [7] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *31st Annual International Symposium on Computer Architecture, 2004. Proceedings.*, pages 64–75, June 2004.
- [8] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable soc. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 174.1, Washington, DC, USA, 2003. IEEE Computer Society.
- [9] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 184–185, 592, February 2005.
- [10] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. *SIGOPS Oper. Syst. Rev.*, 40(5):325–335, 2006.
- [11] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, 2007.
- [12] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.