A Balanced Programming Model for Emerging Heterogeneous Multicore Systems

Wei Liu, Brian Lewis, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Sai Luo, Bratin Saha

Intel Corporation

{wei.w.liu, brian.t.lewis, xiaocheng.zhou, hu.tiger.chen, ying.gao, shoumeng.yan, sai.luo, bratin.saha}@intel.com

Abstract

Computer systems are moving towards a heterogeneous architecture with a combination of one or more CPUs and one or more accelerator processors. Such heterogeneous systems pose a new challenge to the parallel programming community. Languages such as OpenCL and CUDA provide a program environment for such systems. However, they focus on data parallel programming where the majority of computation is carried out by the accelerators. Our view is that, in the future, accelerator processors will be tightly coupled with the CPUs, be available in different system architectures (e.g., integrated and discrete), and systems will be dynamically reconfigurable. In this paper we advocate a *balanced* programming model where computation is balanced between the CPU and its accelerators. This model supports sharing virtual memory between the CPU and the accelerator processors so the same data structures can be manipulated by both sides. It also supports task-parallel as well as data-parallel programming, fine-grained synchronization, thread scheduling, and load balancing. This model not only leverages the computational capability of CPUs, but also allows dynamic system reconfiguration, and supports different platform configurations. To help demonstrate the practicality of our programming model, we present performance results for a preliminary implementation on a computer system with an Intel® Core™ i7 processor and a discrete Larrabee processor. These results show that the model's most performance-critical part, its shared virtual memory implementation, simplifies programming without hurting performance.

1. Introduction

Computer systems are moving towards a heterogeneous architecture with a combination of CPUs and accelerator processors. Historically, CPUs often focused on general purpose computation while the accelerators were specialized to speed up fixed functions such as graphics, audio/video encoding/decoding, encryption [1], and packet processing. Today, however, systems are increasingly being designed to include general-purpose, programmable accelerators like GPGPUs [2] and Intel's Larrabee [3]. The accelerators are used to accelerate non-graphical computation in areas such as fluid dynamics, computational biology, and game physics. How to best program these heterogeneous systems remains a challenge for the parallel programming community.

Most accelerators are currently programmed using a language like CUDA [4] or OpenCL [5]. These languages are effective at solving a wide range of problems although their current emphasis is on data-parallel computation since that is a good match for most contemporary accelerators. A number of trends are emerging, however, that suggest changes in the design of programming models appropriate for emerging heterogeneous systems:

First, computing systems are being designed with integrated accelerators. Communication between a CPU and an accelerator integrated onto the same die as in the upcoming Intel Sandy Bridge [6] and AMD's Fusion processors [7] can be an order of magnitude faster than with a discrete part. Furthermore, integrated devices can share physical memory with the CPU (making copying data between the processors unnecessary) and benefit from hardware-provided coherency support. They also may be able to share locking structures like mutexes. All of these make practical acceleration of fine-grained computation such as SSL encryption.

Some computing systems are also being designed that combine integrated and discrete accelerators. For example, a system might have both a discrete Larrabee processor as well as an integrated GPU. Computation can be done on the higher-performance Larrabee device when sufficient power is available, or on the integrated GPU when power is limited (e.g., when running on batteries). Systems using ATI's Hybrid CrossFireX technology [8] provide similar capabilities. We believe a heterogeneous programming model should support dynamic reconfiguration so work can be scheduled on either the CPU, integrated, or discrete accelerators.

In addition, accelerators like Larrabee and NVIDIA's next generation Fermi processors [9] are significantly more capable than the first-generation GPGPUs. They

have, for example, large virtual address spaces, unified memory, and atomic instructions. Among other things, these features make them better at supporting task-based parallel programming.

Finally, recent CPU designs are manycore processors with wide vector instruction sets such as Intel AVX [10]. This makes the CPUs excellent at data- as well as task-parallelism. We believe demand is increasing for support of both task-parallelism and data-parallelism in a heterogeneous programming model. Programming languages like OpenCL need to better leverage the substantial computational capability of these CPUs and improve how they balance a program's work among the CPU and the accelerators. For example, they need fine-grained synchronization models that fully support CPUs (e.g., OpenCL host threads) as well as accelerators, and they must ensure that API calls made by multiple CPU threads are thread-safe.

In this paper, we propose a balanced programming model for these emerging heterogeneous systems. We describe our programming model as *balanced*, by which we mean that: 1) It balances work among the available CPU and accelerator processors according to workload, processor capabilities, and programmer direction. 2) It provides virtual memory sharing and global, fine-grained synchronization primitives since we believe these are required for effectively making full use of accelerators and CPUs, as well as for allowing applications to be split across the different processors. We are currently designing and implementing this model as an extension of the shared virtual memory system described in [11].

This paper makes the following contributions:

- It describes trends emerging in the design of heterogeneous systems that indicate a need for improvements in current programming models to support those systems.
- It presents a new balanced programming model that leverages the computational power of modern CPUs and accelerators and supports shared virtual memory between them. It also allows dynamic reconfiguration and supports different platform configurations.
- It presents performance results for several workloads using shared memory in our preliminary implementation on a real (not simulated) computer system with an Intel® Core™ i7 processor and a discrete Larrabee processor.

2. Goals for our programming model

Our goal is to simplify programming and to increase the range of applications that run efficiently on emerging heterogeneous systems. To achieve that, we envision a programming environment that dynamically balances fine-grained computation between the CPU and accelerators. We argue that a programming model for future heterogeneous systems should include several features that are missing or not well supported by existing languages like CUDA and OpenCL.

Balanced use of both CPUs and accelerators

Given the trend towards integrating cores onto the same die as CPU cores, it is increasingly important to use the computational capabilities of both in a better way.

In today's OpenCL 1.0 model, CPUs are used in two ways. The first way is to use a CPU to do setup and supervisory work such as initialization or task management. This non-compute intensive use of a CPU may work well on a platform where the accelerator dominates computation, and especially when the CPU-accelerator communication latency is high, but it underutilizes the CPU, particularly in platforms with an integrated accelerator. The second way is to use CPUs as OpenCL devices on which kernels can be queued for execution. However, the capability of CPUs is still not fully utilized since kernels are limited in the operations they can do. For example, kernels cannot do dynamic memory allocation. In addition, OpenCL always requires a host thread that occupies CPU resources. Overall, these problems make programming more difficult.

To overcome these limitations, our balanced programming model uses the computational capability of both the CPU and accelerators. Its dynamic scheduling support tries to make the best use of available processor capacities and the particular capabilities of each kind of processor. It does load balancing and attempts to schedule work on CPUs and accelerators that are the best match for the work required. This enables new applications, ones that are not well supported by OpenCL today, for example, a stream-oriented application which the CPU produces data that the accelerator consumes in a pipeline fashion.

Support for shared virtual memory

We argue that supporting shared virtual memory among CPU and accelerator cores is essential for effectively using heterogeneous systems, since it dramatically simplifies dividing up applications across the different types of cores. This makes it much easier to use the particular capabilities of each CPU and accelerators, and makes it possible to balance the load on the system. In addition, shared virtual memory significantly eases programming. It is tedious and error-prone for programmers to explicitly marshal, copy, and unmarshal data between different processors; we want to directly share the same data structures, including ones that contain pointers. Moreover,

shared virtual memory enables concurrent shared memory accesses from both CPUs and accelerators. Finally, we want to support partially-shared virtual memory to allow non-shared data to be stored privately in unshared portions of the address space to reduce coherence traffic.

OpenCL currently supports data sharing with explicit read and write commands that transfer data between a host CPU and a device. It supports sharing only "flat" data structures such as arrays that do not contain embedded pointers. If a data structure such as a tree of objects is passed to, e.g., a physics engine on a GPU, that tree must be marshaled into a flat representation in a memory buffer, written to the GPU, and then unmarshaled into a new data structure on the GPU. This requires multiple data representations and much error-prone programming.

Support changes to platform configurations

In OpenCL, the programmer must create command queues for the different devices and explicitly enter kernels onto those queues. Changes to the number of devices or their configuration typically require changing the OpenCL program.

In our view, it shouldn't be necessary to rewrite a program in order to run it on a system with a different version of an accelerator processor, or on a system with an integrated accelerator instead of a discrete one. This requires a dynamically-reconfigurable programming model and runtime. In addition, the model should view accelerators as optional in the sense that they can be turned on or off on demand. The model should support dynamic scheduling to balance work among the CPU and available accelerators. This flexibility should also allow systems to better deal with devices failing during execution.

Fine-grained synchronization and task parallelism

OpenCL 1.0 provides an event model that can be used, with careful programming, to ensure that commands and memory updates are synchronized among devices sharing the same context. There is no support, however, for fine-grained synchronizing between host threads executing on CPUs and kernels executing on devices. Besides events, OpenCL also provides work-item barriers. These barriers ensure all work items in a work group must execute the barrier before any continue on, but there is no mechanism to synchronize between different work groups.

We believe a heterogeneous programming model should provide global, fine-grained synchronization primitives. These primitives include mutexes, condition variables, and barriers that are shared by the CPU and accelerator cores. Based on their experience trying to use a GPU to accelerate database query processing, Kaldewey et al [12] note that such global synchronization support is missing in current GPU programming environments. In addition, many existing and new parallel programs are naturally structured using task parallel model and fine-grained synchronization. This makes support for task-based parallelism necessary.

3. A Case study: programming on a discrete and integrated GPU system

We are currently developing and prototyping our heterogeneous programming environment. It is an extension of the shared virtual memory system [11] that is fully implemented and has been successfully used to ease the acceleration of such applications as the Bullet [13] physics engine running with Larrabee. Its virtual memory sharing and synchronization APIs form a basis for our programming model. At this time, we are enhancing our environment to support platforms with integrated accelerators such as ones with an integrated GPU.

3.1. Implementation overview

The key challenge for designing a system like ours comes from the heterogeneity. CPUs and accelerators usually have different ISAs and run different operating systems. Some accelerators don't have a full OS and some have none. In our design, we chose to create a daemon thread on the CPU and one on each accelerator. The daemon threads communicate to support functionality such as memory sharing and synchronization.

Compiler and code generation

We depend on the programmer adding annotations to their C or C++ code to identify code that should be run on an accelerator. The attribute, *__attribute (<accelerator>)*, marks functions that should be executed on a specific accelerator. By default, the compiler generates CPU code. Accelerator code is only generated if programmers request it, in which case the compiler generates both accelerator and CPU versions of the code. It is the compiler's responsibility to optimize the code for the CPU and the accelerator since accelerators usually have different instruction sets than CPUs. Note that it is the runtime that decides which version, the CPU's or the accelerator's, to actually execute. This is based on various runtime situations such as the presence of the accelerator, how busy it is, and the load balance.

The CPU binary contains code for all functions while each accelerator binary contains just the functions that execute on that accelerator. Our runtime library has CPU and accelerator components that are linked with the application binaries to build the executables. When the CPU binary starts executing, it calls a runtime function that loads the accelerator executable. The CPU and accelera-

tor executables create the daemon threads used to communicate between CPUs and accelerators.

Efficiently sharing virtual memory

Virtual memory sharing enables seamless data sharing between CPUs and accelerators. It avoids explicit data movement among threads, which greatly enhances programmability. However, a poor implementation may degrade system performance. In this section, we describe our techniques and design choices that significantly reduce performance loss.

We use release consistency in our model for several reasons. First, it improves efficiency on certain platform configurations. For example, on discrete accelerators, local updates do not have to be transferred to the other processors until a release point, which makes better use of the limited bandwidth. Second, many applications have clear acquire and release points. In general, the points when computation transfers between a CPU and an accelerator are such acquire or release points. For example, in a CPU-GPU platform, it is common for the CPU to set up a data structure, release it to the GPU for processing, and then afterwards acquire back the resulting data structure. Finally, the programming language community has settled on release consistency for the coherency model, as in the Java memory model [14].

To share memory between CPUs and accelerators, our runtime uses a common communication buffer between the daemon threads. This common buffer is typically smaller than the size of the partially-shared virtual memory area, and is implemented differently for discrete and integrated accelerators. For example, we use the PCI aperture as the common buffer on Larrabee, and we use a block of dedicated and pinned physical memory on an integrated Intel Integrated Graphics GPU. During initialization, we set up a task queue, a message queue, and copy buffers in the common buffer and map it into the user space of the application. When we need to copy pages from, for example, the CPU to an accelerator, the runtime copies the pages into the copy buffers and tags the buffers with the virtual address and the process identifier. On the accelerator side, the daemon thread copies the contents of the buffers into its address space by using the virtual address tag. Copies from the accelerator to the CPU are done in a similar way. This two-step approach allows asynchronous execution on both sides and user-level communication, which is vastly more efficient than going through the OS kernel. We further optimize coherence traffic by sending only differences, not whole pages.

Ownership rights as hints

We use ownership rights in our shared memory model to help optimize coherence traffic. A CPU or an accelerator can claim ownership of a particular region of shared space at various points during execution. By knowing who owns a virtual memory region, the implementation can reduce otherwise necessary coherence traffic. For example, when the CPU sets up a data structure, it can claim exclusive ownership of the data. As a result, that shared memory does not have to be kept coherent on accelerators. There is no need to send snoops to accelerators to monitor the data structure. In addition, having multiple shared virtual memory regions often avoids false sharing.

We implemented ownership rights in our runtime independent of the platform configuration. For each shared memory region, there is metadata to identify the memory pages that belong to it. The ownership rights work as hints in the sense that when an ownership violation happens, ownership will be automatically promoted to a shared state by both sides.

Remote (between-processor) calls and synchronization

When a CPU function (i.e. a non-annotated function) calls an annotated accelerator function, it starts a remote call from the CPU to the accelerator. Our language rules ensure that any function involved in remote calls is annotated by users so that the compiler can generate correct accelerator-specific code for them. We implement remote calls using a combination of compiler and runtime techniques. When compiling annotated functions, the compiler inserts a call to register the function addresses at program startup into a jump table that contains <*func_name, func_addr*> for each annotated function.

For annotated functions called remotely (not from the same accelerator), the compiler generated code uses the jump table to look up the function's address. It then packs the arguments into an argument buffer in the shared space, followed by a dispatch call to the accelerator passing the target function address and argument buffer pointer. For a remote call to the CPU, the process is similar.

The runtime also provides global synchronization operations such as mutexes and barriers to allow synchronization between CPU and accelerators.

3.2. Performance results

We evaluated our programming environment's performance on a system with an Intel® Core™ i7 processor running Windows Vista and a discrete Larrabee processor running the Larrabee Micro OS. A similar system was demonstrated at the 2009 Intel Developer Forum. These two processors are connected through PCI-Express 2.0. We used a number of non-graphical workloads to measure performance including a Black-Scholes financial workload, an Equake earthquake modeling program, and

a Canny edge detector. All workloads were written using our programming constructs and compiled with our tools.

Figure 1 shows how well the different workloads perform using our environment compared to an implementation using a device driver and programmer-written, hand-optimized marshal/unmarshal code. Each workload is split between the CPU and Larrabee cores with compute-intensive portions executed on Larrabee. The baseline used one CPU core and one Larrabee core. The graphs show system performance with one CPU core and a varying number of Larrabee cores. Performance was collected for three cases: 1) *No Share* where virtual memory isn't shared between the CPU and Larrabee, and data structures are explicitly copied using the device driver. 2) *Share w/ Ownership* where virtual memory sharing is enabled and optimized with ownership rights. 3) *Share w/o Ownership* where sharing is enabled but ownership rights are not used.

The results show our *Share w/ Ownership* performance is comparable to the *No Share* device driver case across all workloads. This is true across three workloads with different scaling characteristics: for example, Canny does not scale well due to the large amount of data transferred between the CPU and Larrabee. Note that ownership hints help by up to 13%. Note also that the gap between *No Share* and *Share w/ Ownership* tends to grow when the number of Larrabee cores increases beyond 4 cores. This is largely due to the increased overhead of some memory protection and page fault handling primitives we use to implement sharing. We are developing new optimizations to reduce this overhead. We expect an integrated GPU to eliminate much of the data transfer, and are experimenting with this configuration.

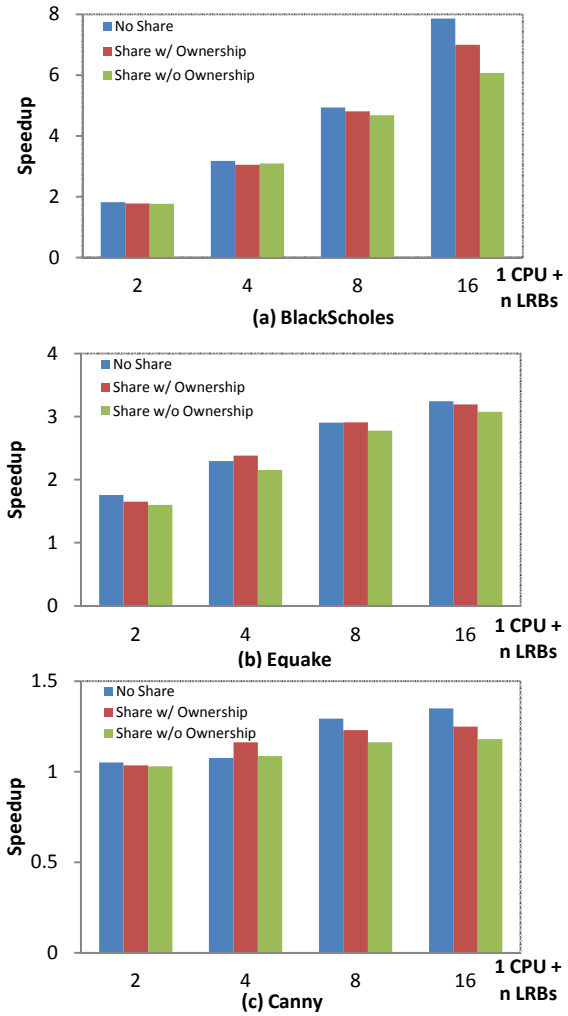

Figure 1 Overall performance comparison

4. Related work

The most closely related programming models are OpenCL [5] and CUDA [4]. However, our work differs from them in several ways. First, we better leverage the CPU's computational capability and better balance work between the CPU and accelerators. Second, our system allows dynamic changes to platform configuration and supports different platform configurations. Finally, our shared virtual memory model eases data sharing between the CPU and accelerators.

Sequoia [15] is a portable, low-level programming language for developing parallel programs that are memory-hierarchy aware. It has constructs to allocate and transfer data in systems with multiple distinct memory spaces. Sequoia programs have been run on PC clusters as well as workstations with multiple Cell processors. While C-like, Sequoia is relatively restricted. Its tasks operate in a private address space and can only access their arguments and local variables. Data movement is limited to task calls and the copy operator. Only C scalars and arrays of scalars may be transferred, not pointer-containing data structures. On the other hand, programmers can write multiple implementations for a task, and select which to use for each call. Our programming model supports only CPU or accelerator versions of procedures and the runtime selects where accelerator-annotated procedures are actually run.

The MAGMA project [16] is developing a dense linear algebra library for heterogeneous platforms. It splits algorithms into tasks in a DAG form and schedules "critical tasks" to the CPU. MAGMA is a linear algebra library similar to LAPACK that dynamically balances work among CPUs and GPUs, but it does not provide a general-purpose programming model for a wide range of applications. It also does not support virtual memory sharing.

Ct [17] is a programming model focusing on throughput oriented applications. It supports data-parallelism over large structures representing vectors, hierarchies, hash tables, etc. Unlike our system, it still requires explicit data movement between the CPU and the accelerator Ct spaces. Moreover, our system has better support for task parallelism such as fine-grained synchronization.

The Cell processor [18] is another heterogeneous platform. The PPE (akin to a CPU) hosts operating systems and serves as a controller to establish a runtime environment for SPE (akin to an accelerator) programs. However, in our model, the work is balanced between CPUs and accelerators. In addition, Cell programming involves explicit DMA between the PPE and SPE while our model supports virtual memory sharing.

5. Conclusions

Systems from netbooks to servers are increasingly being designed as heterogeneous computing platforms that have a combination of CPUs and accelerator processors. While languages like OpenCL and CUDA support programming data-parallel applications well on most contemporary accelerators, they don't fully address the capabilities of emerging processors. They don't support balancing work between CPUs and accelerators, and they don't support sharing virtual memory so that pointer-containing data structures can be shared seamlessly. In this paper, we propose a heterogeneous programming model that balances computation between all available processors. Moreover, the model supports virtual shared memory across CPUs and accelerators. It also supports both task-parallel and data-parallel programming with fine-grained synchronization, thread scheduling, and load balancing. Our model not only naturally leverages the substantial computational abilities of today's CPU's, but it also supports different platform configurations and allows dynamic reconfiguration. We implemented an initial prototype of our model on an Intel® Core™ i7-Larrabee platform and show that it eases programming while providing comparable performance.

6. References

[1] Sun, *Sun Web Server Encryption Solution*, found at http://www.sun.com/servers/coolthreads/sun_web_encryption/.

[2] Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C., and Lefohn, A. 2004. *GPGPU: General Purpose Computation on Graphics Hardware.* SIGGRAPH 2004.

[3] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. 2008. *Larrabee: A Many-Core x86 Architecture for Visual Computing*. ACM Trans. Graph. 27, 3 (Aug. 2008), 1-15.

[4] NVIDIA, *CUDA Programming Environment,* see www.nvidia.com/object/cuda_what_is.html.

[5] *OpenCL 1.0 Specification*, Document Revision 48, at http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf.

[6] Intel, *Next Generation Processors, in Manufacturing, Chip design Expertise Driving Innovation and Integration, Historic Change to Computers*, Sept 2009. http://www.intc.com/releasedetail.cfm?ReleaseID=410888.

[7] Advanced Micro Devices, *AMD CPU Roadmap*, http://developer.amd.com/assets/Develop_Brighton_Justin_Boggs-1.pdf.

[8] Advanced Micro Devices, *ATI Hybrid Graphics Technology*, http://www.amd.com/us/Documents/45689-A-ATI-Hybrid-Gfx-Brochure.pdf.

[9] NVIDIA, *Fermi Compute Architecture White Paper*, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[10] Firasta, N., Buxton, M., Jinbo, P., Nasri, K., and Kuo, S. *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency,* White Paper, http://software.intel.com/en-us/articles/intel-avx-new-frontiers-in-performance-improvements-and-energy-efficiency/.

[11] Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R., and Mendelson, A. *Programming Model for a Heterogeneous X86 Platform*. In Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, June 15-21, 2009.

[12] Kaldewey, T., Hagen, J., Di Blas, A., and Sedlar, E. *Parallel Search on Video Cards*. Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar '09), March 2009.

[13] Bullet Physics, at http://bulletphysics.org/wordpress/.

[14] Manson, J., Pugh, W., and Adve, S. V. *The Java Memory Model*. *SIGPLAN Notices* 40, 1 (Jan. 2005).

[15] Fatahalian, K., Knight, T. J., Houston, M., Erez, M., Horn, D. R., Leem, L., Park, J. Y., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. *Sequoia: Programming the memory hierarchy*. In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing.

[16] Tomov, S., Dongarra, J., Baboulin, M. *Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems, University of Tennessee Computer Science Technical Report, UT-CS-08-632 (also LAPACK Working Note 210)*, October 17, 2008

[17] Ghuloum, A., Smith, T., Wu, G., Zhou, X., Fang, J., Guo, P., Rajagopalan, M., Chen, Y., and Chen, B. *Future-Proof Data Parallel Algorithms and Software on Intel® Multi-Core Architecture*, Intel Technology Journal 11(4), Nov. 2007.

[18] Gschwind, M., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y., Yamakazi, T. *Synergistic Processing in Cell's Multicore Architecture.* IEEE Micro, April 2006.