Scientific Programming 18 (2010) 1–33
DOI 10.3233/SPR-2009-0296
IOS Press

State-of-the-art in heterogeneous computing

Andre R. Brodtkorb [a,*], Christopher Dyken [a], Trond R. Hagen [a], Jon M. Hjelmervik [a] and Olaf O. Storaasli [b]

[a] *SINTEF ICT, Department of Applied Mathematics, Blindern, Oslo, Norway*
E-mails: {Andre.Brodtkorb, Christopher.Dyken, Trond.R.Hagen, Jon.M.Hjelmervik}@sintef.no
[b] *Oak Ridge National Laboratory, Future Technologies Group, Oak Ridge, TN, USA*
E-mail: Olaf@ornl.gov

Abstract. Node level heterogeneous architectures have become attractive during the last decade for several reasons: compared to traditional symmetric CPUs, they offer high peak performance and are energy and/or cost efficient. With the increase of fine-grained parallelism in high-performance computing, as well as the introduction of parallelism in workstations, there is an acute need for a good overview and understanding of these architectures. We give an overview of the state-of-the-art in heterogeneous computing, focusing on three commonly found architectures: the Cell Broadband Engine Architecture, graphics processing units (GPUs), and field programmable gate arrays (FPGAs). We present a review of hardware, available software tools, and an overview of state-of-the-art techniques and algorithms. Furthermore, we present a qualitative and quantitative comparison of the architectures, and give our view on the future of heterogeneous computing.

Keywords: Power-efficient architectures, parallel computer architecture, stream or vector architectures, energy and power consumption, microprocessor performance

1. Introduction

The goal of this article is to provide an overview of node-level heterogeneous computing, including hardware, software tools and state-of-the-art algorithms. Heterogeneous computing refers to the use of different processing cores to maximize performance, and we focus on the Cell Broadband Engine Architecture (CBEA) and CPUs in combination with either graphics processing units (GPUs) or field programmable gate arrays (FPGAs). These new architectures break with the traditional evolutionary processor design path, and pose new challenges and opportunities in high-performance computing.

Processor design has always been a rapidly evolving research field. The first generation of digital computers were designed using analog vacuum tubes or relays and complex wiring, such as the Atanasoff–Berry Computer [69]. The second generation came with the digital transistor, invented in 1947 by Bardeen, Brattain and Shockley, for which they were awarded the 1956 Nobel prize in physics. The transistor dramatically reduced space requirements and increased the speed of logic gates, making computers smaller and more power efficient. Noyce and Kilby independently invented the integrated circuit in 1958, leading to further reductions in power and space required for third generation computers in the early 1960-ies. The integrated circuit rapidly led to the invention of the microprocessor by Intel engineers Hoff, Faggin and Mazor [15] in 1968, the TMS1802NC from Texas Instruments [30], and the Central Air Data Computer CPU by Holt and Geller [83] working for AiResearch. The trend to decrease space and power requirements continues even today, with smaller feature sizes with every new production technique. As size and power consumption per logic gate have been reduced, there has been a proportional growth in computational power. The two main contributors are the number of transistors and the frequency they run at.

In 1965, Moore predicted that the number of transistors inexpensively placed on a single chip would double every two years [122], a trend followed for over thirty years [121]. With a state-of-the-art 32 nm production process, it is now possible, though not inexpensive, to place 16 *billion* transistors on a single chip [102, p. 92], and there are currently no signs suggesting that this exponential growth will stop.

Traditionally, the processor frequency closely follows Moore's law. However, physical constraints have

*Corresponding author: André R. Brodtkorb, SINTEF ICT, Department of Applied Mathematics, P.O. Box 124, Blindern, N-0314 Oslo, Norway. E-mail: Andre.Brodtkorb@sintef.no.

stopped and even slightly reversed this exponential frequency race. A key physical constraint is power density, often referred to as the *power wall*. Kogge et al. [102, p. 88] state the relationship for power density as

$$P = C\rho f V_{dd}^2, \tag{1}$$

where P is the power density in watts per unit area, C is the total capacitance, ρ is the transistor density, f is the processor frequency, and V_{dd} is the supply voltage. The frequency and supply voltage are related, as higher supply voltages allow transistors to charge and discharge more rapidly, enabling higher clock frequencies. It should be noted that the formula ignores leakage power, which typically amounts to 30–40% of the total power consumption [102, p. 93]. The power density in processors has already surpassed that of a hot plate, and is approaching the physical limit of what silicon can withstand with current cooling techniques. Continuing to ride the frequency wave would require new cooling techniques, for example liquid nitrogen or hydrogen, or new materials such as carbon nanotubes. Unfortunately, such solutions are currently prohibitively expensive.

The combination of Moore's law and the power wall restrains processor design. The frequency cannot be increased with today's technology, so performance is primarily boosted by increased transistor count. The most transistor-rich CPU yet, the Intel Itanium Tukwila [158], uses two *billion* transistors. It is increasingly difficult to make good use of that many transistors. It is the trade-off between performance gain and development cost that has evolved processors into their current design, and most transistors are currently devoted to huge caches and complex logic for instruction level parallelism (ILP). Increasing the cache size or introducing more ILP yields too little performance gain compared to the development costs.

The "rule of thumb" interpretation of (1) is that if you decrease the voltage and frequency by one percent, the power density is decreased by three percent, and the performance by 0.66%. Thus, dual-core designs running at 85% of the frequency with 85% of the supply voltage offer 180% better performance than single-core designs, yet consume approximately the same power. Consequently, major silicon vendors now spend their transistor budget on symmetric multi-core processors for the mass market. This evolutionary path might suffice for two, four and perhaps eight cores, as users might run that many programs simultaneously. However, in the foreseeable future we will most likely get hundreds of cores. This is a major issue: if silicon vendors and application developers cannot give better performance to users with new hardware, the whole hardware and software market will go from selling new products, to simply maintaining existing product lines [14].

Todays multi-core CPUs spend most of their transistors on logic and cache, with a lot of power spent on non-computational units. Heterogeneous architectures offer an alternative to this strategy, with traditional multi-core architectures in combination with accelerator cores. Accelerator cores are designed to maximize performance, given a fixed power or transistor budget. This typically implies that accelerator cores use fewer transistors and run at lower frequencies than traditional CPUs. Complex functionality is also sacrificed, disabling their ability to run operating systems, and they are typically managed by traditional cores to offload resource-intensive operations.

Algorithms such as finite-state machines and other intrinsically serial algorithms are most suitable for single-core CPUs running at high frequencies. Embarrassingly parallel algorithms such as Monte Carlo simulations, on the other hand, benefit greatly from many accelerator cores running at a lower frequency. Most applications consist of a mixture of such serial and parallel tasks, and will ultimately perform best on heterogeneous architectures. The optimal type and composition of processors, however, will vary from one application to another [13,80].

With the recent emphasis on green computing, it becomes essential to use all possible resources at every clock cycle, and the notion of *green algorithms* is on the doorstep. Both academia and industry realize that serial performance has reached its zenith, leading to an increased focus on new algorithms that can benefit from parallel and heterogeneous architectures. Further insight into these different architectures, and their implications on algorithm performance, is essential in algorithm design and for application developers to bridge the gap between peak performance and experienced performance. The field of heterogeneous computing covers a large variety of architectures and application areas, and there is currently no unified theory to encompass it all. Fair comparisons of the architectures are therefore difficult. Our goal is to give thorough comparisons of the CBEA and CPUs in combination with GPUs or FPGAs, and to contribute new insight into the future of heterogeneous computing.

We begin with an overview of traditional hardware and software in Section 2, followed by a review of the

three heterogeneous architectures in Section 3. In Section 4 we discuss programming concepts, followed by an overview of state-of-the-art algorithms and applications in Section 5. Finally, we conclude the article in Section 6, including our views on future trends.

2. Traditional hardware and software

In this article, we use the term chip to denote a single physical package, and core to denote a physical processing core. The term processor has, unfortunately, become ambiguous, and may refer to a chip, a core or even a virtual core. We begin by describing parallelism and memory hierarchies, and then give an overview of relevant CPU architectural details, and shortfalls revealing why they cannot meet future performance requirements. Finally, we give a short summary of programming concepts and languages that form a basis of knowledge for heterogeneous computing.

There are multiple layers of parallelism exposed in modern hardware, including the following:

Multi-chip parallelism is having several physical processor chips in a single computer sharing resources, in particular system memory, through which relatively inexpensive communication is done.

Multi-core parallelism is similar to multi-chip parallelism, but the processor cores are contained within a single chip, thus letting the cores share resources like on-chip cache. This makes communication even less expensive.

Multi-context (thread) parallelism is exposed within a single core when it can switch between multiple execution contexts with little or no overhead. Each context requires a separate register file and program counter in hardware.

Instruction parallelism is when a processor can execute more than one instruction in parallel, using multiple instruction units.

Current CPUs use several of these techniques to decrease cycles per instruction and to hide memory latency. Examples include hardware pipelining, where multiple instructions are simultaneously in the pipeline; vector parallelism, where an instruction is replicated over an array of arithmetic units; and superscalar processors, where multiple instructions are dispatched to different execution units either automatically in hardware, or explicitly using very long instruction words (VLIWs). Techniques that target the memory latencies are also plentiful. Out-of-order execution reorders an instruction stream to minimize the number of processor stalls caused by latencies of data dependencies. Hardware multi-threading lets a set of execution contexts share the same execution units. The CPU instantaneously switches between these contexts when memory requests stall, decreasing the impact of latencies. This should not be confused with software threads, where the different execution contexts typically are stored in main memory. Such techniques are usually combined, and make program execution complex and hard to predict.

Traditionally, floating-point operations were considered expensive, while retrieving data was practically free. However, this conventional wisdom has changed [13], and memory access has grown to become the limiting factor in most applications. Data is moved in or out of the processor using a limited number of pins running at a limited frequency, referred to as the *von Neumann bottleneck* [17]. In addition, there is a significant latency to initiate data transfers. From 1980 to 1996, the gap between processor and memory performance grew annually by 50% [142]. To bridge this gap, large memory hierarchies have been developed, addressing both the von Neumann bottleneck and memory latency by copying requested data to faster memory. This enables rapid access to recently used data, increasing performance for applications with regular memory access. In the following, we classify a memory hierarchy according to latency:

Registers are the closest memory units in a processor core and operate at the same speed as the computational units.

Local store memory or scratchpad memory, resembles a programmable cache with explicit data movement. It has a latency of tens of clock cycles.

Caches have rapidly grown in size and number. On modern CPUs, there is typically a hierarchy of two or three layers that operate with a latency of tens of clock cycles. Off-chip caches, also found in some computers, have a latency somewhere between on-chip cache and main memory.

Main memory has a latency of hundreds of clock cycles, and is limited in bandwidth by the von Neumann bottleneck.

The recent end of the frequency race, mentioned in the introduction, has caused the relative increase in latency to halt, a positive side effect. The von Neumann bottleneck, however, continues to be a burden. It can

be alleviated, somewhat, by using non-uniform memory access (NUMA) on shared-memory machines. On NUMA machines, memory is physically distributed between cores, and the access time depends on where the memory is physically located relative to the core. All major silicon vendors have now started producing chip-level NUMA processors, making placement of data in the memory hierarchy important. Another way to address the von Neumann bottleneck, is simply to increase cache size until it exceeds the working set size. However, increasing the cache size directly corresponds to increased latency, only countered by smaller feature sizes [48, pp. 16–17]. Hardware caches are very efficient for programs with predictable and regular memory access, but cannot speed up programs with random memory access. Even worse, using caches for random access applications can degrade performance, as the cache transfers full cache lines across the memory bus when only a single element is needed without reuse [48, pp. 34–35, 47].

Data movement is not only a limiting factor in terms of execution times, but also seriously affects the energy consumption [102, pp. 225–227]. The energy to move one bit between two levels in the memory hierarchy is around 1–3 pico joule [102, p. 211], and energy consumed by memory operations is therefore becoming a major constraint for large clusters.

Parallel computing has a considerable history in high-performance computing. One of the key challenges is to design efficient software development methodologies for these architectures. The traditional methodologies mainly focus on task-parallelism and data-parallelism, and we emphasize that the two methodologies are not mutually exclusive.

Task-parallel methodology roughly views the problem as a set of tasks with clearly defined communication patterns and dependencies. A representative example is pipelining. Fortran M [61] was one of the early approaches to task-parallelism in Fortran and MPI [68] is another prime example of traditional task-parallelism.

Data-parallel methodology, on the other hand, roughly views the problem as a set of operations carried out on arrays of data in a relatively uniform fashion. Early approaches use a global view methodology, where there is one conceptual thread of execution, and parallel statements such as FORALL enable parallelism. Fortran D [101] and Vienna Fortran [187] are early examples of the global view methodology. High Performance Fortran (HPF) [97,151] followed and HPF+ [25] further introduced task-parallelism, as well as addressing various issues in HPF. OpenMP [39] is another example that exposes global view parallelism, and supports both task- and data-parallel programming in C, C++ and Fortran. Co-Array Fortran [125], Unified Parallel C [173] and Titanium [79] on the other hand, expose a Partitioned Global Address Space (PGAS) programming model, where each conceptual thread of execution has private memory in addition to memory shared among threads.

The problem of extending C or Fortran is that modern programming language features, such as object orientation, possibilities for generic programming and garbage collection, are missing. However, there are examples of parallel programming languages with these features. Titanium [79] is based on Java without dynamic threads, Chapel [38] exposes object orientation and generic programming techniques, and X10 [150] integrates reiterated concepts for parallel programming alongside modern programming language features. These traditional and modern languages form the basis for new languages, where existing ideas, for example, from the glory days of vector machines, are brought back to life for heterogeneous architectures.

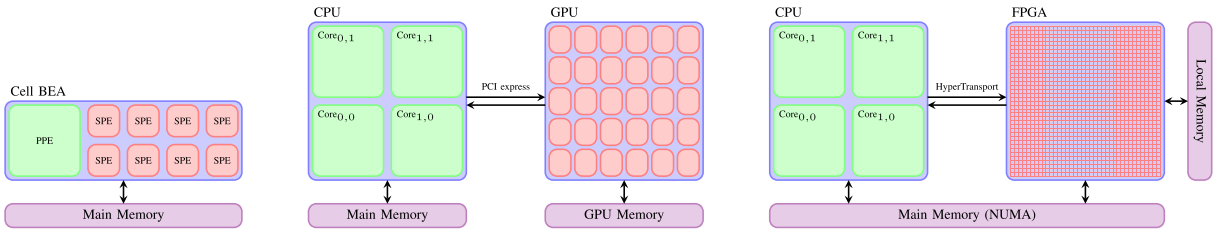
3. Heterogeneous architectures

The introduction described the current tendency to increase performance by parallelism instead of clock frequency. Our focus is on parallelism within a single node, where instruction-level parallelism is nearly fully exploited [77, pp. 154–192]. This means that increased performance must come from multi-chip, multi-core or multi-context parallelism. Flynn's taxonomy [58] defines four levels of parallelism in hardware:

(1) single instruction single data (SISD),
(2) single instruction multiple data (SIMD),
(3) multiple instruction single data (MISD) and
(4) multiple instruction multiple data (MIMD).

In addition, two subdivisions of MIMD are single program multiple data (SPMD), and multiple program multiple data (MPMD). We use these terms to describe the architectures.

The single-chip CBEA, illustrated in Fig. 1(a), consists of a traditional CPU core and eight SIMD accelerator cores. It is a very flexible architecture, where each core can run separate programs in MPMD fashion and communicate through a fast on-chip bus. Its main design criteria has been to maximise performance



(a) CBEA (b) CPU in combination with GPU (c) CPU in combination with FPGA

Fig. 1. Schematic of heterogeneous architectures we focus on: The Cell Broadband Engine is a heterogeneous chip (a), a CPU in combination with a GPU is a heterogeneous system (b), and a CPU in combination with an FPGA is also a heterogeneous system (c). The GPU is connected to the CPU via the PCI express bus, and some FPGAs are socket compatible with AMD and Intel processors.

whilst consuming a minimum of power. Figure 1(b) shows a GPU with 30 highly multi-threaded SIMD accelerator cores in combination with a standard multi-core CPU. The GPU has a vastly superior bandwidth and computational performance, and is optimized for running SPMD programs with little or no synchronization. It is designed for high-performance graphics, where throughput of data is key. Finally, Fig. 1(c) shows an FPGA consisting of an array of logic blocks in combination with a standard multi-core CPU. FPGAs can also incorporate regular CPU cores on-chip, making it a heterogeneous chip by itself. FPGAs can be viewed as user-defined application-specific integrated circuits (ASICs) that are reconfigurable. They offer fully deterministic performance and are designed for high throughput, for example, in telecommunication applications.

In the following, we give a more detailed overview of the hardware of these three heterogeneous architectures, and finally sum up with a discussion, comparing essential properties such as required level of parallelism, communication possibilities, performance and cost.

3.1. Graphics processing unit architecture

The GPU was traditionally designed for use in computer games, where 2D images of 3D triangles and other geometric objects are *rendered*. Each element in the output image is referred to as a pixel, and the GPU uses a set of processors to compute the color of such pixels in parallel. Recent GPUs are more general, with rendering only as a special case. The theoretical peak performance of GPUs is now close to three teraflops, making them attractive for high-performance computing. However, the downside is that GPUs typically reside on the PCI express bus. Second generation PCI express $\times 16$ allows 8 GB/s data transfers between CPU and GPU memory, where 5.2 GB/s is attainable on benchmarks.

A GPU is a symmetric multi-core processor that is exclusively accessed and controlled by the CPU, making the two a heterogeneous system. The GPU operates asynchronously from the CPU, enabling concurrent execution and memory transfer. AMD, Intel and NVIDIA are the three major GPU vendors, where AMD and NVIDIA dominate the high-performance gaming market. Intel, however, has disclosed plans to release a high-performance gaming GPU called Larrabee [155]. In the following, we give a thorough overview of the NVIDIA GT200 [132] and AMD RV770 [6] architectures, followed by a brief description of the upcoming Intel Larrabee. The first two are conceptually quite similar, with highly multi-threaded SIMD cores, whereas the Larrabee consists of fewer, yet more complex, SIMD cores.

Current NVIDIA hardware is based on the GT200 architecture [132], shown in Fig. 2. The GT200 architecture is typically programmed using CUDA [133] (see Section 4.2), which exposes an SPMD programming model using a large number of threads organized into *blocks*. All blocks run the same program, referred to as a *kernel*, and threads within one block can synchronize and communicate using *shared memory*, a kind of local store memory. Communication between blocks, however, is limited to atomic operations on global memory.

The blocks are automatically split into *warps*, consisting of 32 threads. The blocks are scheduled to the streaming multiprocessors (SMs) at runtime, and each warp is executed in SIMD fashion. This is done by issuing the same instruction through four consecutive runs on the eight scalar processors (SPs). Divergent code flow between threads is handled in hardware by automatic thread masking and serialization, and thus, divergence within a warp reduces performance, while divergence between warps has no im-

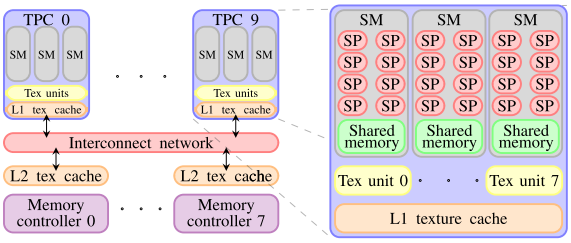


Fig. 2. NVIDIA GT200 GPU architecture. The abbreviations have the following meaning: TPC – texture processing cluster; SM – streaming multiprocessor; Tex unit – texture unit; Tex cache – texture cache; SP – scalar processor.

pact on execution speed. In addition to eight scalar processors, the streaming multiprocessor also contains a double precision unit, two special function units, 16 KiB of shared memory, 8 KiB constant memory cache and 16,384 32-bit registers. Each scalar processor is a fully pipelined arithmetic-logic unit capable of integer and single precision floating point operations, while the special function unit contains four arithmetic-logic units, mainly used for vertex attribute interpolation in graphics and in calculating transcendentals. The streaming multiprocessor is a dual-issue processor, where the scalar processors and special function unit operate independently.

All the threads within a block have, as mentioned, access to the same shared memory. The shared memory is organized into 16 memory banks that can be accessed every other clock cycle. As one warp uses four clock cycles to complete, it can issue two memory requests per run, one for each *half warp*. For full speed, it is vital that each thread in a half warp exclusively accesses one memory bank. Access to shared memory within a warp is automatically synchronized, and barriers are used to synchronize within a block.

The streaming multiprocessors are designed to keep many active warps in flight. It can switch between these warps without any overhead, which is used to hide instruction and memory latencies. A single multiprocessor executes all warps within a block, but it can also run multiple blocks. The number of blocks each multiprocessor can run is dictated by the register and shared memory use of their respective warps, which cannot exceed the physically available resources. The ratio of active warps to the maximum number of supported warps is referred to as *occupancy*, which is a measure indicating how well the streaming multiprocessor may hide latencies.

The GT200 has eight 64-bit memory controllers, providing an aggregate 512-bit memory interface to main GPU memory. It can either be accessed from the streaming multiprocessors through the texture units that use the texture cache, or directly as *global memory*. Textures are a computer graphics data-format concept, and can be though of as a read-only 2D image. Texture access is thus optimized for 2D access, and the cache holds a small 2D neighbourhood, in contrast to the linear caching of traditional CPUs. The texture units can perform simple filtering operations on texture data, such as interpolation between colors. Three and three streaming multiprocessors are grouped into *texture processing clusters*, that share eight such texture units and a single L1 texture cache. Global memory access has a high latency and is optimized for linear access. Full bandwidth is achieved when the memory requests can be *coalesced* into the read or write of a full memory segment. In general, this requires that all threads within one half-warp access the same 128-bit segment in global memory. When only partial segments are accessed, the GT200 can reduce the segment size, reducing wasted bandwidth. Per-thread arrays, called *local memory*, are also available. In contrast to previous array concepts on GPUs, these arrays can be accessed dynamically, thus not limited to compile-time constants. However, local memory resides in an auto-coalesced global memory space, and has the latency of global memory. The threads can also access main CPU memory via *zero-copy*, where data is moved directly over the PCI express bus to the streaming multiprocessor. A great benefit of zero-copy is that it is independent of main GPU memory, thus increasing total bandwidth to the streaming multiprocessors.

NVIDIA have also recently released specifications for their upcoming GT300 architecture, codenamed Fermi [131]. Fermi is based around the same concepts as the GT200, with some major improvements. First of all, the number of Scalar Processors has roughly doubled, from 240 to 512. The double precision performance has also improved dramatically, now running at half the speed of single precision. All vital parts of memory are also protected by ECC, and the new architecture has cache hierarchy with 16 or 32 KiB L1 data cache per Streaming Multiprocessor, and a 768 KiB L2 cache shared between all Streaming Multiprocessors. The memory space is also unified, so that *shared memory* and *global memory* use the same address space, thus enabling execution of C++ code directly on the GPU. The new chip also supports concurrent kernel execution, where up-to 16 independent kernels can execute simultaneously. The Fermi is currently not available in stores, but is expected to appear shortly.

The current generation of AMD FireStream cards is based on the RV770 [6] architecture, shown in Fig. 3. Equivalent to the NVIDIA concept of a grid of blocks, it employs an SPMD model over a *grid* of *groups*. All groups run the same kernel, and threads within a group can communicate and synchronize using local data store. Thus, a group resembles the concept of blocks on NVIDIA hardware.

Each group is an array of threads with fully independent code-flow, and groups are scheduled to *SIMD engines* at runtime. The SIMD engine is the basic core of the RV770, containing 16 shader processing units (SPUs), 16 KiB of local data share, an undisclosed number of 32-bit registers, 4 texture units and an L1 texture cache. Groups are automatically split up into *wavefronts* of 64 threads, and the wavefronts are executed in SIMD fashion by four consecutive runs of the same instruction on the 16 shader processing units. The hardware uses serialization and masking to handle divergent code flow between threads, making divergence within wavefronts impact performance, whereas divergence between wavefronts runs at full speed. As such, wavefronts are equivalent to the concept of warps on NVIDIA hardware. However, unlike the scalar design of the NVIDIA stream processor, the shader processing unit is super-scalar and introduces instruction level parallelism with five single precision units that are programmed using very long instruction words (VLIW). The super-scalar design is also reflected in registers, which use four-component data types. The fifth unit also handles transcendental and double-precision arithmetic.

All threads within a group have access to the same *local data share*, which is somewhat similar to the shared memory of NVIDIA. Data is allocated per thread, which all threads within the group can access. However, threads can only write to their own local data share. Synchronization is done using memory barriers, as on NVIDIA hardware. The RV770 also exposes shared registers, which are persistent between kernel invocations. These registers are shared between all wavefronts processed by a SIMD engine, in which thread i in wavefront A can share data with thread i in wavefront B, but not with thread j. Thus, any operation on shared registers is atomic.

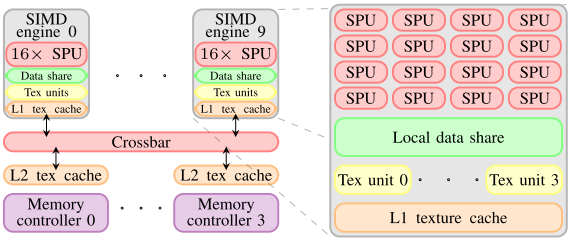


Fig. 3. AMD RV770 GPU architecture. The abbreviations have the following meaning: SPU – shader processing unit; Tex unit – texture unit; Tex cache – texture cache.

Thread grids are processed in either pixel shader or compute mode. The pixel shader mode is primarily for graphics, and restricts features to that of the R6XX architecture without local data share. Using the compute shader mode, output is required to be a *global buffer*. Global buffers are accessed directly by the threads, allowing arbitrary read and write operations. As global buffers are not cached, *burst writes* are used for consecutive addresses in a fashion similar to coalesced memory write on NVIDIA hardware. On current hardware, however, global buffers are inferior to regular buffers with respect to performance. Threads can also access part of main CPU memory; the GPU driver reserves a small memory segment that the SIMD engines can access directly over the PCI express bus. The RV770 also contains instruction and constant memory caches, an L2 texture cache, and a global data share that are shared between the SIMD engines. The global data share enable the SIMD engines to share data, but is currently not exposed to the programmer.

The upcoming Larrabee [155] GPU from Intel can be considered a hybrid between a multi-core CPU and a GPU. The Larrabee architecture is based on many simple in-order CPU cores that run an extended version of the $\times$86 instruction set. Each core is based on the Pentium chip with an additional 16-wide SIMD unit which executes integer, single-precision, or double-precision instructions. The core is dual-issue, where the scalar and SIMD units can operate simultaneously, and it supports four threads of execution. The Larrabee also features a coherent L2 cache shared among all the cores, which is divided into local subsets of 256 KiB per core. Cores can communicate and share data using on-chip communication, consisting of a 1024-bit bi-directional ring network. A striking design choice is the lack of a hardware rasterizer, which forces the Larrabee to implement rasterization in software. The first Larrabee GPUs are expected to arrive in 2010.

The architectures from NVIDIA and AMD are, as mentioned earlier, conceptually similar. They operate with the same execution model, but have differences in their hardware setups. AMD has twice the SIMD width of NVIDIA hardware, runs at about half the frequency, and employs a superscalar architecture. However, they both feature similar peak performance

numbers at around one teraflops in single precision. The memory bandwidth is also strikingly similar, just surpassing 100 GB/s, even though they have different hardware setups. When utilizing texture sampling units, an interesting note is that NVIDIA has three execution units per texture sampler, whereas AMD has four. This can impact performance of algorithms with heavy use of texture sampling.

3.2. Cell BEA

The CBEA is a heterogeneous processor with a traditional CPU core and eight accelerator cores on the same chip, as shown in Fig. 4. It is used in the first petaflops supercomputer, called the Roadrunner [21]. The Roadrunner is the worlds fastest computer on the June 2009 Top500 list [171], and the seven most energy-efficient supercomputers on this list use the CBEA as the main processor [172]. Commercially, it is available as 1U form factor or blade servers, as PCI express plug-in cards, and in the PlayStation 3 gaming console. The state-of-the-art server solutions use the PowerXCell 8i version of the CBEA. For example, the Roadrunner supercomputer consists of two 3.2 GHz CBEAs per dual-core 1.8 GHz Opteron, in which the CBEAs contribute to 96% of the peak performance [167]. The PCI express plug-in cards also use the PowerXCell 8i processor, but are primarily intended for development and prototyping. The CBEA in the PlayStation 3 is inexpensive due to its high production numbers, but it is a different chip than the PowerXCell 8i. It offers low double precision performance and a very limited amount of main memory. Furthermore, one of the accelerator cores is disabled in hardware to increase production yields, and another accelerator core is reserved for the Hypervisor virtualization layer when running Linux. We focus on the PowerXCell 8i implementation that consists of the Power Processing Element (PPE), eight Synergistic Processing Elements (SPEs) and the Element Interconnect Bus (EIB).

The PPE is a traditional processor that uses the Power instruction set. It contains a 512 KiB L2 cache, an in-order dual-issue RISC core with two-way hardware multi-threading, and a VMX [87] engine for SIMD instructions. Compared to modern CPUs, the PPE is a stripped-down core with focus on power efficiency. It is capable of outputting one fused multiply-add in double precision, or one SIMD single precision fused multiply-add per clock cycle. Running at 3.2 GHz, this yields a peak performance of 25.6 or 6.4 gigaflops in single and double precision, respectively.

The SPE consists of a memory flow controller (MFC) and a Synergistic Processing Unit (SPU). The SPU is a dual-issue in-order SIMD core with 128 registers of 128 bit each and a 256 KiB local store. The SPU has a vector arithmetic unit similar to VMX that operates on 128-bit wide vectors composed of eight 16-bit integers, four 32-bit integers, four single-precision, or two double-precision floating point numbers. The even pipeline of the dual-issue core handles arithmetic instructions simultaneously as the odd pipeline handles load, store, and control. The SPU lacks a dynamic branch predictor, but exposes hint-for-branch instructions. Without branch hints, branches are assumed not to be taken. At 3.2 GHz, each SPU is capable of 25.6 or 12.8 gigaflops in single and double precision, respectively.

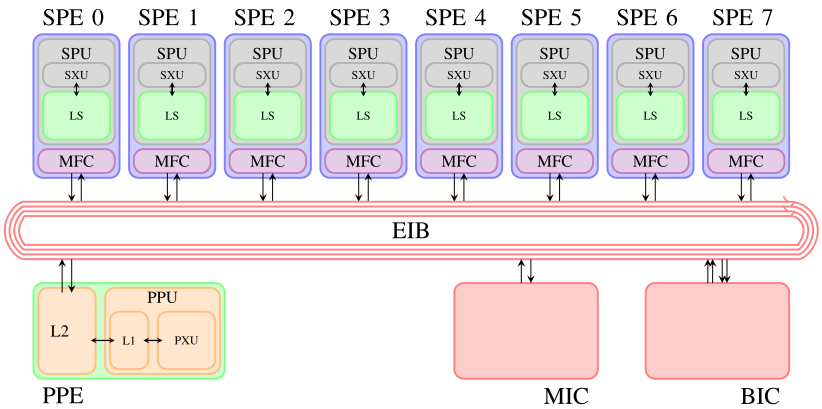


Fig. 4. PowerXCell 8i architecture. The abbreviations have the following meaning: SPE – synergistic processing element; SPU – synergistic processing unit; SXU – synergistic execution unit; LS – local store; MFC – memory flow controller; EIB – element interconnect bus; PPE – power processing element; PPU – power processing unit; PXU – power execution unit; MIC – memory interface controller; BIC – bus interface controller.

The SPU uses the local store similarly to an L2 cache, and the MFC uses DMA requests to move data between the local store and main memory. Because the MFC operates asynchronously from the SPU, computations can take place simultaneously as the MFC is moving data. The MFC requires 16-byte alignment of source and destination data addresses and transfers multiples of 128 bytes. Smaller transfers must be placed in the *preferred slot* within the 128-byte segment, where unwanted data is automatically discarded. The MFC also has mailboxes for sending 32-bit messages through the element interconnect bus, typically used for fast communication between the SPEs and the PPE. Each SPE has one outbound mailbox, one outbound interrupt mailbox, and one inbound mailbox capable of holding four messages.

At the heart of the CBEA is the EIB, a ring bus with two rings in each direction that connects the PPE, the SPEs, the Memory Interface Controller (MIC) and the Bus Interface Controller (BIC). The MIC handles main memory and the BIC handles system components such as the HyperTransport bus. The EIB is capable of moving an aggregate 128 byte $\times$ 1.6 GHz $=$ 204.8 GB/s, and 197 GB/s has been demonstrated [41].

3.3. Field programmable gate array architecture

The first commercially viable FPGA was developed by Xilinx co-founder Ross Freeman in 1984, and he was entered into the National Inventors Hall of Fame for this accomplishment in 2006. FPGAs were initially used for discrete logic, but have expanded their application areas to signal processing, high-performance embedded computing, and recently as accelerators for high-performance computing. Vendors such as Cray, Convey, SGI, HP and SRC all offer such high-performance solutions. While early FPGAs had sufficient capability to be well suited for special-purpose computing, their application for general-purpose computing was initially restricted to a first-generation of low-end reconfigurable supercomputers, for which PCI interfaces were sufficient for CPU communication. However, this situation has recently changed with the adoption of high-speed IO standards, such as QuickPath Interconnect and HyperTransport. The major FPGA vendors, Altera and Xilinx have collaborated with companies such as DRC Computer, Xtreme Data, Convey and Cray, who offer FPGA boards that are socket compatible with Intel or AMD processors, using the same high-speed bus. These boards also offer on-board memory, improving total bandwidth.

An FPGA is a set of configurable logic blocks, digital signal processor blocks, and optional traditional CPU cores that are all connected via an extremely flexible interconnect. This interconnect is reconfigurable, and is used to tailor applications to FPGAs. When configured, FPGAs function just like application specific integrated circuits (ASICs). We focus on the popular Xilinx Virtex-5 LX330, consisting of 51,840 *slices* and 192 digital signal processing slices that can perform floating point multiply-add and accumulate. Heterogeneous FPGAs, such as the Virtex-5 FX200T, offer up-to two traditional 32-bit PowerPCs cores on the same chip. Each slice of a Virtex-5 consists of four 6-input look-up tables and four flip-flops, as shown in Fig. 5. The multiplexers are dynamically set at runtime, whereas the flip-flops are statically set at configuration. Two slices form a configurable logic block, and programmable interconnects route signals between blocks. Configuring the interconnect is part of FPGA programming, and can be viewed as creating a datapath through the FPGA. This can make FPGAs fully deterministic when it comes to performance. A key element to obtaining high-performance on FPGAs is to use as many slices as possible for parallel computation. This can be achieved by pipelining the blocks, trading latency for throughput; by data-parallelism, where data-paths are replicated; or a combination of both. As floating point and double-precision applications rapidly exhausted the number of slices available on early FPGAs, they were often avoided for high-precision calculations. However, this situation

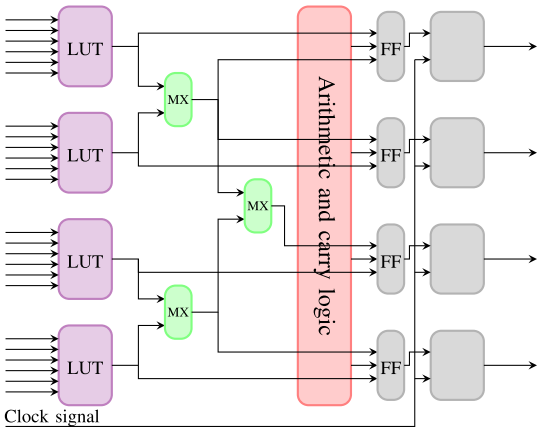


Fig. 5. Simplified diagram of a Virtex-5 FPGA slice. The abbreviations have the following meaning: LUT – look-up table; MX – multiplexer; FF – flip–flop.

has changed for current FPGAs which have sufficient logic to compute thousands of adds, or about 80 64-bit multiplies per clock cycle. The conceptually simple design of FPGAs make them extremely versatile and power-efficient, and they offer high computational performance.

3.4. Performance, power consumption and cost

Due to the lack of a unified theory for comparing different heterogeneous architectures, it is difficult to give a fair comparison. However, in Table 1, we summarize qualitative and quantitative characteristics of the architectures, giving an overview of required level of parallelism, performance and cost. The performance numbers per watt and per dollar are measured in single precision. However, comparing the different metrics will ultimately be a snapshot of continuous development, and may change over time. There are also differences between different hardware models for each architecture: in particular, there are less expensive versions of all architectures, especially for the CPU and GPU, that can benefit from mass production. However, we wish to indicate trends, and argue that the differences we distinguish are representable, even though there are variations among models and in time.

The SIMD units of the state-of-the-art Intel Core i7-965 Quad Extreme processor is capable of 16 single precision multiply-add instructions per clock cycle at 3.2 GHz, that is, 102.4 gigaflops single precision performance, and half of that in double precision. The i7-965 has a thermal design power rating of 130 watt, indicating around 0.8 single precision gigaflops per watt. It has a manufacturer suggested retail price of 999 USD, giving around 100 megaflops per USD.

The NVIDIA Tesla C1060 has 30 streaming multiprocessors that each can execute eight single precision multiply-adds on the scalar processors, and either eight single precision multiplies or one double precision multiply-add on the special function unit. This totals to 720 single precision operations per clock cycle at 1.3 GHz, yielding 936 gigaflops single precision performance, or one twelfth of that in double precision. The C1060 draws a maximum of 187.8 watt, yielding around 5 gigaflops per watt. The manufacturer suggested retail price of 1699 USD gives around 550 megaflops per USD. The C1060 has 4 GiB of GDDR3 memory, and with a 512-bit memory interface at 800 MHz, the peak memory bandwidth is 102 GB/s.

The up-coming Fermi architecture from NVIDIA doubles the number of stream processors in the chip, and increases the double precision performance to half the speed of single precision. Assuming a similar clock frequency as the C1060, one can expect roughly double the performance in single precision, and a dramatic improvement in double precision performance. The Fermi will be available with up-to 6 GB memory on a 384-bit wide GDDR5 bus. This roughly gives a

Table 1

Summary of architectural properties. The numbers are of current hardware, and are reported per physical chip. The CPU is an Intel Core i7-965 Quad Extreme, the AMD GPU is the FireStream 9270, the NVIDIA GPU is the Tesla C1060, the CBEA is the IBM QS22 blade, and the FPGA is the Virtex-6 SX475T

	CPU	GPU (AMD/NVIDIA)	CBEA	FPGA
Composition	Symmetric multicore	Symmetric multicore	Heterogeneous multicore	Heterogeneous multicore
Full cores	4	–	1	2
Accelerator cores	0	10/30	8	2016 DSP slices
Intercore communication	Cache	None	Mailboxes	Explicit wiring
SIMD width	4	64/32	4	Configurable
Additional parallelism	ILP	VLIW/Dual-issue	Dual-issue	Configurable
Float operations per cycle	16	1600/720	36	520
Frequency (GHz)	3.2	0.75/1.3	3.2	$<$0.55
Single precision gigaflops	102.4	1200/936	230.4	550
Double:single precision performance	1:2	1:5/1:12	$\sim$1:2	$\sim$1:4
Gigaflops/watt	0.8	5.5/5	2.5	13.7
Megaflops/USD	70	800/550	$>$46	138
Accelerator bandwidth (GB/s)	N/A	109/102	204.8	N/A
Main memory bandwidth (GB/s)	25.6	8	25.6	6.4
Maximum memory size (GiB)	24	2/4	16	System dependent
ECC support	Yes	No	Yes	Yes

50% bandwidth increase compared to the C1060, again assuming a similar frequency.

The AMD FireStream 9270 contains 160 shader processing units. Assuming full utilization of instruction level parallelism, each shader processing unit can execute five multiply-add instructions per clock cycle resulting in a total of 1600 concurrent operations. Running at 750 MHz this yields a performance of 1.2 teraflops in single precision, and one fifth of that in double precision. The card draws a maximum of 220 watt, implying approximately 5.5 gigaflops per watt. It has a manufacturer suggested retail price of 1499 USD, resulting in 800 megaflops per USD. The 9270 has 2 GiB of GDDR5 memory on a 256-bit memory bus. Running at 850 MHz, the bus provides a peak internal memory bandwidth of 109 GB/s.

AMD recently released their first DirectX11 compatible GPUs, based on the R800 [7] architecture. The R800 is quite similar to the R700 series GPUs, even though the exact architectural details are unpublished. The Radeon HD 5870 uses the R800 architecture, with a total of 320 shader processing units. This chip runs at 850 MHz, and has a peak performance of 2.7 teraflops, again assuming full utilization of instruction level parallelism. The new card has 1 GB memory on a 256-bit wide GDDR5 bus. Running at 1200 MHz this results in a bandwidth of 153 GB/s.

The IBM QS22 blade [88] consists of two PowerXCell 8i CBEAs connected through a HyperTransport bus. Each of the CBEAs is capable of 230 gigaflops in single precision, and slightly less than half that in double precision. The CBEA consumes 90 watt, yielding 2.5 gigaflops per watt. The QS22 blade with 8 GiB memory has a manufacturer suggested retail price of 9995 USD, implying 46 megaflops per USD. However, this rather pessimistic figure includes the cost of the whole blade server.

On FPGAs, floating point operations are often traded away due to their real-estate demands. However, if one were to implement multiply-add's throughout the FPGA fabric, the 32-bit floating point performance attains 230 gigaflops for the Virtex-5 LX330, 210 gigaflops for the SX240T and 550 gigaflops for the Virtex-6 SX475 [161,162]. The SX475T performance is due to the 2016 DSP slices, twice that of the SX240T and ten times as many as the LX330. Double precision multiply-add units consume four times the real-estate, and twice the memory, implying about one fourth double precision performance. FPGA cost is about twice processor cost. The LX330, SX240T and SX475T cost about 1700, 2000 and 4000 USD, respectively, but deliver approximately 17, 18 and 14 gigaflops per watt. An example of an FPGA board that is socket compatible with AMD Opteron CPUs is the Accelium AC2030. It can access main memory at 6.4 GB/s, has three HyperTransport links running at 3.2 GB/s, and has 4.5 GiB on-board RAM with a bandwidth of 9.6 GB/s. Equipped with the LX330, the board uses 40 watts.

3.5. Numerical compliance and error resiliency

Floating point calculations most often contain errors, albeit sometimes small. The attained accuracy is a product of the inherent correctness of the numerical algorithm, the precision of input data, and the precision of intermediate arithmetic operations. Examples abound where double precision is insufficient to yield accurate results, including Gaussian elimination, where even small perturbations can yield large errors. Surprising is Rump's example, reexamined for IEEE 754 by Loh and Walster [108], which converges towards a completely wrong result when precision is increased. Thus, designing numerically stable algorithms, such as convex combinations of B-spline algorithms, is essential for high-precision results. Floating point operations and storage of any precision have rounding errors, regardless of algorithmic properties, and even the order of computations significantly impacts accuracy [78].

Using high precision storage and operations is not necessarily the best choice, as the accuracy of an algorithm might be less than single or double precision [54, 71,167]. Using the lowest possible precision in such cases has a two-fold benefit. Take the example of double versus single precision, where single precision is sufficiently accurate. Both storage and bandwidth requirements are halved, as single precision consumes half the memory. Further, single precision increases compute performance by a factor 2–12, as shown in Table 1. Single precision operations can also be used to produce high precision results using mixed precision algorithms. Mixed precision algorithms obtain high-precision results, for example, by using lower precision intermediate calculations followed by high-precision corrections [62,63,164].

GPUs have historically received scepticism for the lack of double precision and IEEE-754 floating point compliance. However, current GPUs and the CBEA both support double precision arithmetic and conform to the floating point standard in the same way as SIMD units in CPUs do [73]. FPGAs can ultimately be tuned

to your needs. Numerical codes executing on these architectures today typically yield bit-identical results, assuming the same order of operations, and any discrepancies are within the floating point standard.

Spontaneous bit errors are less predictable. Computer chips are so small that they are susceptible to cosmic radiation, build-up of static charge on transistors, and other causes which can flip a single bit. Several technologies can detect and correct such errors. One is ECC memory where a parity bit is used to detect and correct bit errors in a word. Others perform ECC logic in software [170], or duplicate computations and check for consistency. There is also research into algorithmic resiliency [102, p. 231], possibly the best solution. Using a few extra computations, such as computing the residual, the validity of results can be determined.

4. Programming concepts

Heterogeneous architectures pose new programming difficulties compared to existing serial and parallel platforms. There are two main ways of attacking these difficulties: inventing new, or adapting existing languages and concepts. Parallel and high-performance computing both form a basis of knowledge for heterogeneous computing. However, heterogeneous architectures are not only parallel, but also quite different from traditional CPU cores, being less robust for non-optimal code. This is particularly challenging for high-level approaches, as these require new compiler transformation and optimization rules. Porting applications to heterogeneous architectures thus often requires complete algorithm redesign, as some programming patterns, which in principle are trivial, require great care for efficient implementation on a heterogeneous platform. As Chamberlain et al. [37] note, heterogeneous architectures most often require a set of different programming languages and concepts, which is both complex from a programmers perspective, as well as prone to unforeseen errors. Furthermore, a programming language that maps well to the underlying architecture is only one part of a productive programming environment. Support tools like profilers and debuggers are just as important as the properties of the language.

In this section, we describe programming languages, compilers, and accompanying tools for GPUs, the CBEA, and FPGAs, and summarize with a discussion that compares fundamental characteristics such as type of abstraction, memory model, programming model and so on. We also report our subjective opinion on ease of use and maturity.

4.1. Multi-architectural languages

OpenCL [98] is a recent standard, ratified by the Khronos Group, for programming heterogeneous computers. Khronos mostly consists of hardware and software companies within parallel computing, graphics, mobile, entertainment, and multimedia industries, and has a highly commercial incentive, creating open standards such as OpenGL that will yield greater business opportunities for its members.

Khronos began working with OpenCL in June 2008, in response to an Apple proposal. The Khronos ratification board received the standard draft in October, ratifying it in December 2008. Most major hardware and many major software vendors are on the ratification board, giving confidence that OpenCL will experience the same kind of success as OpenMP. Apple included OpenCL in Mac OS X Snow Leopard, and both NVIDIA and AMD have released beta-stage compilers. It is also probable that CBEA compilers will appear, as the CBEA team from IBM has participated in the standard. Support for FPGAs, however, is unclear at this time.

OpenCL consists of a programming language for accelerator cores and a set of platform API calls to manage the OpenCL programs and devices. The programming language is based on C99 [94], with a philosophy and syntax reminiscent of CUDA (see Section 4.2) using SPMD kernels for the data-parallel programming model. Task-parallelism is also supported with single-threaded kernels, expressing parallelism using 2–16 wide vector data types and multiple tasks. Kernel execution is managed by command queues, which support out-of-order execution, enabling relatively fine-grained task-parallelism. Out-of-order tasks are synchronized by barriers or by explicitly specifying dependencies. Synchronization between command queues is also explicit. The standard defines requirements for OpenCL compliance in a similar manner as OpenGL, and extensively defines floating-point requirements. Optional extensions are also defined, including double precision and atomic functions.

RapidMind [113] is a high-level C++ abstraction to multi-core CPU, GPU and CBEA programming. It originated from the research group around Sh [114] in 2004, and is a now a commercial product. It has a common abstraction to architecture-specific back-ends. Low-level optimization and load balancing is handled by the back-ends, allowing the programmer to focus on algorithms and high-level optimizations.

They argue that for a given development time, a programmer will create better performing code using the RapidMind compared to lower level tools [148]. RapidMind exposes a streaming model based on arrays, and the programmer writes kernels that operate on these arrays. Their abstraction is implemented as a C++ library and thus requires no new tools or compilers. Debugging is done with traditional tools using a debugging back-end on the CPU, and performance statistics can also be collected.

A more high-level approach is HMPP, the Hybrid Multi-core Parallel Programming environment [29]. Their approach is to annotate C or Fortran source code, similar to OpenMP, to denote the parts of the program that should run on the device. In addition, special pragmas give hints about data movement between the host and device. The HMPP consists of a meta compiler and runtime library, that supports CUDA and CAL-enabled GPUs. A similar approach is taken by the Portland Group with their Accelerator C and Fortran compilers [146], currently supporting CUDA enabled GPUs. Such approaches have the advantage of easy migration of legacy code, yielding good performance for embarrassingly parallel code. However, many algorithms require redesign by experts to fully utilize heterogeneous architectures. Relying on high-level approaches in these cases will thus yield sub-optimal performance.

4.2. Graphics processing unit languages

Initially, GPUs could only be programmed using graphics APIs like OpenGL [137]. General purpose stream-computing was achieved by mapping stream elements to pixels. The obvious drawback was that the developer needed a thorough understanding of computer graphics. As a result, higher-level languages were developed that ran on top of the graphics API, such as Brook [33] and RapidMind [113]. To provide a more direct access to the GPU, AMD has released Stream SDK [8] and NVIDIA has released CUDA [133]. Both Stream SDK and CUDA are tied to their respective vendor's GPUs, which is an advantage when considering performance, but a problem for portability. Direct3D [119] is a graphics API from Microsoft targeting game developers. It standardizes the hardware and is supported by both NVIDIA and AMD. Direct3D 11, which is expected to appear shortly, includes a *compute shader*, whose main goal is to tightly integrate GPU accelerated 3D rendering and computation, such as game physics. The Direct3D compute shader uses ideas similar to NVIDIA CUDA.

The CUDA Toolkit [129] provides the means to program CUDA-enabled NVIDIA GPUs, and is available on Windows, Linux and Mac OS X. CUDA consists of a runtime library and a set of compiler tools, and exposes an SPMD programming model where a large number of threads, grouped into blocks, run the same kernel.

The foundation of the CUDA software stack is CUDA PTX, which defines a virtual machine for parallel thread execution. This provides a stable low-level interface decoupled from the specific target GPU. The set of threads within a block is referred to as a cooperative thread array (CTA) in PTX terms. All threads of a CTA run concurrently, communicate through shared memory, and synchronize using barrier instructions. Multiple CTAs run concurrently, but communication between CTAs is limited to atomic instructions on global memory. Each CTA has a position within the grid, and each thread has a position within a CTA, which threads use to explicitly manage input and output of data.

The CUDA programming language is an extension to C, with some C++-features such as templates and static classes. The new Fermi architecture enables full C++ support, which is expected to appear in CUDA gradually. The GPU is programmed using CUDA kernels. A kernel is a regular function that is compiled into a PTX program, and executed for all threads in a CTA. A kernel can directly access GPU memory and shared memory, and can synchronize execution within a CTA. A kernel is invoked in a CUDA context, and one or more contexts can be bound to a single GPU. Multiple contexts can each be bound to different GPUs as well, however, load balancing and communication between multiple contexts must be explicitly managed by the CPU application. Kernel execution and CPU–GPU memory transfers can run asynchronously, where the order of operations is managed using a concept of streams. There can be many streams within a single GPU context: all operations enqueued into a single stream are executed in-order, whereas the execution order of operations in different streams is undefined. Synchronization events can be inserted into the streams to synchronize them with the CPU, and timing events can be used to profile execution times within a stream.

CUDA exposes two C APIs to manage GPU contexts and kernel execution: the runtime API and the driver API. The runtime API is a C++ abstraction where both GPU and CPU code can be in the same

source files, which are compiled into a single executable. The driver API, on the other hand, is an API that requires explicit handling of low-level details, such as management of contexts and compilation of PTX assembly to GPU binary code. This offers greater flexibility, at the cost of higher code complexity.

There are several approaches to debugging CUDA kernels. NVCC, the compiler driver that compiles CUDA code, can produce CPU emulation code, allowing debugging with any CPU debugger. NVIDIA also provides a beta-release of a CUDA capable GNU debugger, called cuda-gdb, that enables interactive debugging of CUDA kernels running on the GPU, alongside debugging of CPU code. Cuda-gdb can switch between CTAs and threads, step through code at warp granularity, and peek into kernel variables. NVIDIA has also announced an up-coming Visual Studio toolset called Nexus that integrates debugging and performance analysis [130].

CUDA also contains a non-intrusive profiler that can be invoked on an existing binary. The profiler can log kernel launch attributes such as grid-size, kernel resource usage, GPU and driver execution times, memory transfers, performance counters for coalesced and non-coalesced loads and stores, local memory usage, branch divergence, occupancy and warp serialization. NVIDIA also provides the CUDA Visual Profiler, which is a GUI front-end for profiling CUDA applications, and the CUDA Occupancy Calculator for experimenting with how register and shared memory use affects the occupancy on different GPUs.

NVIDIA GPUs native instruction sets are proprietary, but many details have been deciphered through differential analysis of compiled GPU kernels. In particular, decuda [175], a third party disassembler for compiled CUDA kernels, is a valuable tool to analyze and optimize kernels.

The AMD Stream SDK [8], available on Windows and Linux, is the successor of ATI's Close To the Metal initiative from 2006, and provides the means to program AMD GPUs directly. The software stack consists of the AMD Compute Abstraction Layer (CAL), which supports the R6XX-series and newer AMD GPUs, and the high-level Brook+ language. CAL exposes an SPMD programming model, in which a program is executed by every thread in a large grid of threads. Threads are initialized with their grid positions, which can be used for explicit input and output of data. The output of threads can also be defined implicitly by using the pixel shader mode, which can improve performance. Threads are clustered into groups, where threads within a single group can synchronize using barriers and communicate using local data store.

CAL has two major components, the CAL runtime and the CAL compiler. The CAL compiler is a C interface used to compile GPU assembly or CAL intermediate language, a GPU-agnostic assembly language similar to CUDA PTX, into GPU binaries. The CAL runtime is a C-interface reminiscent of the CUDA driver API with respect to programming complexity. It is used to create GPU contexts, load and execute kernels on the GPU, and manage GPU resources. A context is tied to a single GPU, and holds a queue of operations that are asynchronously executed in order. Multiple contexts can be created on a single GPU, and they can share resources. However, resource sharing and load balancing between different GPUs has to be explicitly managed by the application. New features are added to CAL through an extension mechanism. One extension enables resources sharing with DirectX, and another enables performance counters of cache hit rate and SIMD engine idle cycles.

Brook+ is AMD's extension of Brook [33], and is a high-level programming language and runtime, with CPU and CAL back-ends. Brook+ is a higher-level language compared to CUDA. The programmer defines kernels that operate on input and output data streams, and multiple kernels operating on the same streams create an implicit dependency graph. Data is explicitly copied from the CPU to input streams, and from output streams back to the CPU after kernel execution. Copying data into streams and execution of kernels run asynchronous, enabling concurrent CPU computations. Reading data back to the CPU from a stream is blocking by default. Brook+ allows streams and kernels to be assigned to different GPUs, however, streams used by a kernel are required to reside on the GPU running the kernel, i.e., inter-GPU communication has to be explicitly managed by the application.

The basic Brook+ kernel maps input stream elements one-to-one to output stream elements. Reduction kernels reduce the dimensionality of a stream along one axis using an associative and commutative binary operator. Kernels can also have explicit communication patterns, providing random-access gather operations from input streams, scatter write to a single output stream, as well as access to the local data share.

A Brook+ application contains both code running on the CPU, and kernels running on the GPU. The Brook+ kernel language is an extension of C, with syntax to denote streams and keywords to specify kernel and stream attributes. Kernels are processed by the

Brook+ compiler, which produces a set of C++-files that implement the CPU interface of the kernel, and the kernel itself as either CAL intermediate language, Direct3D HLSL, or CPU emulation code. The generated C++-files can then be included into any C++ project.

Both intermediate language code and Brook+ code can be analyzed using the Stream Kernel Analyzer [9]. It is a graphical Windows application that shows the native GPU assembly code, kernel resource use, prediction on whether arithmetic or memory transfer is the bottleneck, and performance estimates for a full range of GPUs.

Both NVIDIA and AMD offer assembly level programming of their GPUs. NVIDIA also offers CUDA and AMD offers Brook+, both based on C to write kernels. However, CUDA offers an abstraction closer to the hardware, whereas Brook+ abstracts away most hardware features. This enables experienced CUDA programmers to write highly tuned code for a specific GPU. Brook+, on the other hand, enables rapid implementation of programs, giving compilers responsibility for optimizations.

4.3. Cell BEA languages

Recall from Section 3.2 that the CBEA consists of one PPE and eight SPEs connected by the Element Interconnect Bus. While regular CPUs come with logic for instruction level parallelism and hardware managed caches, the CBEA focuses on low power consumption and deterministic performance, sacrificing out-of-order execution and dynamic branch prediction. This shifts responsibility for utilizing the SPE hardware resources to the programmer and compiler, and makes execution time fully deterministic as long as memory contention is avoided.

The IBM Cell SDK [89] is the de facto API for programming the CBEA. The PPE is typically programmed using Fortran, C or C++, where the SDK offers function calls to manage the SPEs. These functions include ways of creating, starting, and stopping SPE contexts, communicating with the SPEs, etc. Each SPE can run a separate program and communicate with all other nodes on the EIB, such as the PPE and main memory. This enables techniques such as SPE pipelining. The SPEs are also programmed using Fortran, C or C++, but with certain restrictions; e.g., using `iostream` in C++ is prohibited. The SPE compilers also support SIMD intrinsics, similar to VMX, and intrinsics for managing the memory flow controller. Part of the strength of the SPE is the memory flow controller, which gives the programmer full control over data movement. This enables techniques such as software-managed threads [19], [82, pp. 67–68], which can hide memory latency using the same philosophy as hardware threads. The SDK also includes a software cache that can hide the details of data movement at the cost of a small overhead.

There are currently two compilers for the CBEA architecture, GCC and IBM XL. The latter is often given credit for yielding the best performance for moderately to highly tuned code. To compile and link code for the CBEA, the SPE source code is compiled and linked to an SPE executable, which is then embedded into a PPE object file. This object file is finally linked with the rest of the PPE code to produce a CBEA executable. The limited size of local store is overcome by using manually or automatically generated code overlays for large codes. The code overlay mechanism enables arbitrary large codes to run on an SPE by partitioning the SPE program into segments that each fit in local store. Segments are then transparently transferred from main memory at run-time, however, switching between segments is expensive, and should be avoided.

The CBEA also includes solid support for program debugging, profiling, and analysis, all included in the IBM Cell SDK. The compiler can output static timing analysis information that shows pipeline usage for the SPEs. Profiling information can be collected using OProfile. Hardware counters can be accessed through the Performance Counter Tool, and runtime traces can be generated using the Performance Debugging Tool. Vianney et al. [179] give a step-by-step guide to porting serial applications to the CBEA using these tools, which can be accessed directly, or through the Eclipse-based Visual Performance Analyzer. Further, the IBM Full-System Simulator for the CBEA [144] is a cycle accurate simulator, capable of simulating the PPE and SPE cores, the memory hierarchy, and disk and bus traffic. The simulator can output raw traces and has a graphical user interface to visualize statistics.

Programming with the IBM Cell SDK requires detailed knowledge of the hardware, and thus, there has been research into higher-level abstractions. Currently, there are four main abstractions; OpenMP, MPI, CellSs and Sequoia. The OpenMP approach uses a PPE-centric shared-memory model, where the program runs on the PPE and data-intensive parts are offloaded to the SPEs. The other three approaches, on the other hand, employ an SPE-centric distributed memory model, in which the PPE simply manages the work and executes support functions for the SPEs.

OpenMP [39] is a set of pragmas used to annotate parallel sections of the source code. The compiler uses these annotations to generate parallel binary code. OpenMP is implemented in most modern compilers for multi-core architectures, and the IBM XL includes an implementation that leverages the computational power of the SPEs [52,53,134].

MPI [68] is the de facto API for parallel programming on clusters. It exposes a distributed memory model with explicit communication. Ohara et al. [135] were the first to use the MPI programming model to abstract the CBEA. Currently, there are two main implementations of MPI for the CBEA; one by Kumar et al. [104] and Krishna et al. [103]; and The Cell Messaging Layer [140]. Whilst the former employs a traditional MPI message passing philosophy, the latter focuses on receiver-initiated message passing to reduce latencies. Of the MPI abstractions, the Cell Messaging Layer appears to be the most mature technology.

Cell superscalar (CellSs) [24,143] is a source-to-source compiler for the CBEA. CellSs is similar to OpenMP, as the source code is annotated, but instead of annotating parallel sections, functions that can be offloaded to the SPEs are annotated. These functions are then scheduled to the SPEs according to their dependencies.

Sequoia [55,100] is a programming language for the CBEA and traditional CPU clusters. It targets vertical data movement in the memory hierarchy, such as moving data between main memory and the local store of an SPE. This contrasts the horizontal data movement found in programming models such as MPI. Vertical data movement is exposed to the programmer as a tree of tasks, where the tasks are typically used as wrappers for optimized native functions. An automatic tuning framework for Sequoia also exists [149]. However, it should be noted that there has been no public release of Sequoia since 2007.

4.4. Field programmable gate array languages

The major drawback of FPGAs has been the time and expense to program them. Using FPGAs has clearly been cost-effective for communications, high-performance embedded computing, military, and space applications, but problematical for typical high-performance computing. Not all applications can benefit from FPGAs, and even for those that do, nothing is automatic: obtaining speedups may be time-consuming and arduous work. The ideal candidate for FPGA acceleration contains a single computational kernel that comprises most of the program runtime. This computational kernel should be possible to divide into hundreds of tasks or data-parallel operations. One example of such a program is the Smith–Waterman algorithm for local DNA sequence alignment, which illustrates the potential for FPGA acceleration [160]. As FPGAs were developed by logic designers, they are traditionally programmed using circuit design languages such as VHDL [51] and Verilog [178]. These languages require the knowledge and training of a logic designer, take months to learn and far longer to code efficiently. Even once this skill is acquired, VHDL or Verilog coding is strenuous, taking months to develop early prototypes and often years to perfect and optimize. FPGA code development, unlike high-performance computing compilers, is slowed by the additional steps required to synthesize, place and route the circuit. These steps often take hours, or overnight, to complete. However, once the time is taken to code applications efficiently in VHDL, its FPGA performance is excellent. Since 2000, the severe programming limitation has been tended to by dozens of C-like programming languages such as Mitrion-C [120], Impulse-C [92], System-C [138] and Celoxica [36], and graphical languages such as DSPlogic [49] and Viva [166]. There is also an increasing use of shared libraries offered by Xilinx [183] and OpenFPGA [136].

Starbridge Systems and DSPlogic provide graphical icon-based programming environments. Similar to Labview, Viva allows FPGA users to write and run scientific applications without having to deal with esoteric timing and pinout issues of digital logic that require much attention in VHDL and Verilog. Viva coding is a two-step process: first code is written, debugged and tested using the graphical interface, and then automatic place and route synthesis is performed for the target FPGA system. This graphical coding process alleviates the need to know VHDL or Verilog, while the second step simplifies development, enabling users to focus on developing and debugging their algorithms. Viva has been used at NASA with great success.

An innovative approach for users to program FPGAs without circuit design skills, is provided by Mitrion-C and other “C to gate” languages. Users can program the Mitrion Virtual Processor in Mitrion-C, a C-based programming language with additions for FPGA memory and data access. The first step when using Mitrion-C is, just as for Viva, to design, debug and test the program on a regular computer. The second step involves place and route synthesis, often a time consuming task.

4.5. Discussion

With heterogeneous architectures, care is required by the programmer to fully utilize hardware. One problem in sharing hardware is that execution units may starve each other, for example fighting over the same set of cache lines. Unfortunately, the future holds no promise of an easy way out, so programmers must write algorithms suitable for heterogeneous architectures to achieve scalable performance. Table 2 shows representative languages mentioned in this section, comparing abstraction levels, memory models, and our subjective rating of their ease of use and maturity. In addition, Listing 1 shows actual code for each architecture.

All of the architectures we have described use memory latency hiding techniques. The GPU and CBEA both employ memory parallelism, where multiple outstanding memory requests are possible. On the GPU this is implicit by the use of hardware multi-threading, whereas it is explicitly managed on the CBEA in terms of DMA queues. Furthermore, the CBEA can lessen the effect of latencies by using software threads, or overlapping communication and computation with multi-buffering techniques. The FPGA typically employs pipelined designs in which on-chip RAM is used to store intermediate results. This efficiently limits off-chip communication. In contrast, traditional CPUs use power-hungry caches to automatically increase experienced memory performance. However, as previously noted, automatic caches can also worsen performance for many algorithms.

Traditional CPU cores impose special requirements to alignment when using SIMD instructions. Data typically has to be aligned to quadword boundaries, and full quadwords are loaded into the SIMD registers of the processors. The concept of coalesced and burst memory access on GPUs, and the requirements of alignment on the CBEA is strikingly similar. Thus, for high-performance code, there is little difference in the complexity of efficient memory access.

There is a wide difference between the architectures we have described when it comes to programming complexity and flexibility. All of the architectures may eventually be possible to program using OpenCL, and RapidMind already offers a common platform for the CPU, GPUs and the CBEA. However, the architecture-specific languages expose certain differences. Programming the GPU for general purpose problems has become much easier with CUDA and Brook+. Nevertheless, writing code that fully utilizes the hardware can still be difficult. One remark for GPU programming is that optimization can be strenuous, where slight changes to the source code can have dramatic effects on the execution time. Another issue with

Table 2
Summary of programming languages properties

	OpenMP	MPI	OpenCL	CUDA	Brook+	libspe	VHDL/Verilog	Mitrion-C
Target platform	CPU, CBEA	CPU	CPU, GPU, CBEA	GPU (NV)	GPU (AMD)	CBEA	FPGA	FPGA
Availability	Win, Linux, Mac	Win, Linux, Mac	Win, Linux, Mac	Win, Linux, Mac	Win, Linux	Linux	Win, Linux, Mac	Win, Linux, Mac
Abstraction	Pragmas	API	API	API, compiler	API, compiler	API, compiler	API	Compiler
Host language	C, C++, Fortran	C, C++, Fortran	C	C, C++	C++	C, C++, Fortran	“C-like”	C
Kernel language	—	—	C99-based	C99-based, some C++	C99-based	C, Fortran, almost C++	—	C
Memory model	Shared	Distributed	∼PGAS	∼PGAS	Data streams	∼PGAS	all	all
Data-parallelism	Global view	—	SPMD, SIMD	SPMD	SPMD	MPMD	all	all
Task-parallelism	—	Explicit	Full	Streams	Streams	Explicit	all	all
Ease of use	⋆⋆⋆	⋆	⋆⋆	⋆⋆	⋆⋆	⋆	⋆	⋆
Maturity	⋆⋆⋆	⋆⋆⋆	⋆	⋆⋆⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆

Notes: Under abstraction, “compiler” implies that an extra compiler has to be integrated into the toolchain. VHDL and Verilog are described as “C-like”, even though they capture much more than typical sequential C. Under kernel language, CUDA supports some features as templates and function overloading, and libspe supports full C++ except parts of the STL. Several of the languages have a memory model similar to Partitioned Global Address Space (PGAS). Under task parallelism, “explicit” implies explicit communication, “streams” implies multiple in-order asynchronous streams, and “full” implies out-of-order execution with automatic handling of given dependencies. Ease of use and maturity refers to our subjective opinion on the programming languages.

(a) CPU (OpenMP)

```
void add(float* c, float* a, float* b, int w, int h) {
#pragma omp parallel for
  for (int j=0; j<h; ++j) {
    for (int i=0; i<w; ++i) {
      c[j*h+i] = a[j*h+i] + b[j*h+i];
    }
  }
}
```

(b) FPGA (VHDL)

```
 architecture behave of maxtrix_adder is
   constant w : integer := 10;
   constant h : integer := 10;
   signal a, b : array(0 to w-1, 0 to h-1)
                 of std_logic_vector(7 down to 0);
   signal c : array(0 to w-1, 0 to h-1)
              of std_logic_vector(8 down to 0);
 begin
   w_set : for i in 0 to w-1 generate
   begin
     h_set : for j in 0 to h-1 generate
     begin
       c(i, j) <= a(i,j) + b(i,j);
     end generate h_set;
   end generate w_set;
 end behave;
```

(c) GPU (CUDA)

```
__global__ void addKernel(float* c, float* a, float* b) {
  int i = blockIdx.i*blockDim.i+threadIdx.i;
  int j = blockIdx.j*blockDim.j+threadIdx.j;
  int w = gridDim.i*blockDim.i;
  c[j*w+i] = a[j*w+i] + b[j*w+i];
}
```

(d) CBEA (libspe)

```
int main(unsigned long long speid,
     unsigned long long argp,
     unsigned long long envp) {

  MyAddContext ctx __attribute__((aligned(128)));
  vector float *a, *b, *c;
  int nbytes;

  myDMAReadAsync(&ctx, argp, sizeof(MyAddContext));
  myDMAFlush();

  nbytes = ctx.n*sizeof(float);
  a = (vector float*) malloc_align(nbytes);
  b = (vector float*) malloc_align(nbytes);
  c = (vector float*) malloc_align(nbytes);

  myDMARead(b, ctx.b_addr, nbytes);
  myDMARead(c, ctx.c_addr, nbytes);
  myDMASync();

  for (int i=0; i<ctx.n/4; ++i)
    c[i] = spu_add(a[i], b[i]);

  myDMAWrite(ctx.c_addr, c, nbytes);
   myDMASync();

  return 0;
}
```

Listing 1. Comparison of matrix addition using different programming languages. The CPU shows the explicit double for-loop, where the out-most loop is run in parallel. The FPGA code shows a matrix adder that can add two matrices in each clock cycle. In addition to the code shown, around five lines of CPU code is required to open the FPGA device, load its registers with pointers to input and output data, start execution, and finally to close the device. In addition, the FPGA needs VHDL code that can read and write the matrices directly by using the pointers. The GPU, on the other hand is invoked in parallel over a grid, where each thread within a block has an implicitly given position. Only the GPU code is shown, and around ten lines of CPU code is required to allocate GPU memory, upload data to the GPU, execute the kernel, and read data back to main memory. Finally, the CBEA code shows how DMA requests load data into the local store prior to computation, and write back after completion. Notice that functions that begin with "my" have been renamed for readability. In addition to the SPE code shown, the program requires around 20 lines of code to create a context for each SPE, load the program into the SPEs, set arguments, create a CPU thread for each SPE context, and wait for the SPE programs to complete.

GPUs is that they are best suited for streaming applications, because global communication and synchronization is particularly expensive. The CBEA, on the other hand, offers software and hardware where communication and synchronization is efficient. However, the programmer must explicitly manage scheduling and load balancing for maximum performance. Finally, for efficiency, FPGAs are typically programmed using low level languages that require a detailed knowledge of the hardware design, with an especially time consuming place and route stage which generates a valid design for a target FPGA. Nevertheless, this also implies that FPGAs can be extremely efficient for well-suited algorithms.

5. Algorithms and applications

As noted in the introduction, most algorithms can benefit from heterogeneous architectures, where the application itself determines the best distribution of traditional and accelerator cores. Codes which lack compute intensity and have large I/O requirements might prove challenging, as exemplified by the evaluation of Van Amesfoort et al. [174], where they compare multicore CPU, the CBEA, and the GPU for a data-intensive radio-astronomy application. Large I/O requirements also make assessing performance problematic as individual kernel performance can deviate from overall application performance. Comparing individual perfor-

mance can in general be difficult, as data movement times between host and device memory may vary. For some applications it is valid to assume data resides in device memory, while for others it is not.

Asanovic et al. [13,14] identify 13 distinct computing bottlenecks they term *motifs*. Che et al. [40] have qualitatively examined three of these motifs, dense linear algebra, combinatorial logic, and dynamic programming. Comparing development costs and performance on an FPGA, multi-core CPU and a GPU, they conclude that the FPGA excels at combinatorial logic, where efficient use of bit operations can be used, and GPUs excel at parallel workloads with deterministic memory access. The complex data flow of linear algebra, however, does not adapt well to FPGAs. The same is true for floating point, where fixed point arithmetic is more efficient. GPUs, on the other hand, have no problem with floating point, but perform poorly when there is limited exposed parallelism, and many memory accesses, such as that of the dynamic programming motif. Flynn et al. [59] state that FPGAs benefit most when used with deep pipelines. By storing intermediate data in registers or local memory, the von Neumann bottleneck is efficiently bypassed. However, as Beeckler and Gross [22] note in their work on particle simulations, development on FPGAs can still be costly when using languages such as VHDL. Using higher abstractions to the FPGA, such as Mitrion-C increases productivity dramatically. Mitrion-C code can compute 112 million particles per second with three Xilinx Virtex-4 LX200 FPGAs. FPGAs have also been used for tasks such as genome sequencing, molecular dynamics, weather/climate and linear algebra, with a 10–100 times speedup compared to CPUs [159]. Pico Computing, an FPGA board manufacturer, have also accelerated genome sequencing. They report a speedup of over 5000 times over CPUs using 112 FPGA devices, and the FPGAs consume less than 300 watts of power [44,86]. Leveraging the CBEA for sequence analysis has been explored by Sachdeva et al. [152], where compute intensive functions are performed on the SPEs, and logic kept on the PPE. They conclude that the CBEA is attractive from an energy perspective. On the CBEA based Roadrunner supercomputer [21], Swaminarayam et al. [167] take an existing molecular dynamics code, and restructure it to suit the heterogeneous architecture. They achieve 368 teraflops, which corresponds to 28% of peak performance. In addition to speed increase, their implementation increased load balancing possibilities, as well as freeing up Opteron resources, enabling simultaneously analysis, visualization, or checkpointing.

There has been an exponential interest and use of node-level heterogeneous architectures in recent years. The maturity of heterogeneous computing has in the same time period gone from a proof-of-concept state to industrially interesting technology. With the ever-increasing application areas of heterogeneous architectures, an exhaustive survey is impossible. Therefore, we focus on a select few algorithms and applications that together form a basis for understanding state-of-the-art approaches. Table 3 gives an overview of current approaches to heterogeneous computing.

5.1. Numerical linear algebra

Numerical linear algebra is a main building block in many numerical algorithms, making efficient execution essential. Basic Linear Algebra Subprograms (BLAS) [47] is the de facto API for low-level linear algebra operations. Implementations of BLAS exist for a variety of different architectures, and much work has been done to optimize these due to their frequent use. Both NVIDIA and AMD provide GPU-accelerated BLAS implementations, namely CUBLAS [126] and ACML–GPU [10]. BLAS is divided into three *levels*, where the first two operate on vectors, typically making them bandwidth limited. Utilizing more processor cores on such bandwidth limited problems has no effect as long as the bandwidth remains constant. On the other hand, using the vast bandwidth of accelerators, such as the GPU and CBEA, can increase performance over CPU implementations. FPGAs can also increase performance via pipelined designs where latency is traded for throughput. To take full advantage of the floating point performance of accelerators, applications must exhibit a high ratio of arithmetic to memory operations. Level 3 BLAS operates on matrices, and typically performs $\mathcal{O}(n^3)$ operations on $\mathcal{O}(n^2)$ data elements, thus well suited for heterogeneous architectures.

A key operation to solve dense linear systems is matrix multiplication. Matrix multiplication exhibits high arithmetic density, has regular memory access patterns and control flow, making it one of the very few operations where experienced and theoretical floating-point performance roughly match. Larsen and McAllister [105] developed the first algorithm for matrix multiplication using a graphics API, before GPU floating-point arithmetic were available. Matrix multiplication on GPUs has since been a popular research topic. CUBLAS 2.0 uses the matrix multiplication algorithm developed by Volkov and Demmel [180], optimized not only to hide memory latency between main

Table 3
Summary of state-of-the-art approaches in heterogeneous computing

Application	Approach
Miscellaneous	Use of different programming languages (FPGA) [22]; Qualitative comparison (FPGA, CPU, GPU) [40]; Bioinformatics (CBEA) [152]; Restructuring of code for heterogeneous architectures (CBEA) [167]; Molecular Dynamics (FPGA) [159]
Dense linear algebra	Achieving peak performance (CBEA) [5,41,70]; Matrix multiplication, LU decomposition (FPGA) [185]; Use of registers instead of shared memory (GPU) [180]; Linpack (CBEA) [99]; Mixed precision (FPGA) [164]
Sparse linear algebra	Blocking (CBEA) [182]; Data structures (GPU) [23]
Fast Fourier transform	Auto-tuning (GPU) [124]; Hierarchically decomposed (GPU) [66]; Iterative out-of-place (CBEA) [18]; Communication overheads (CBEA) [184]; Power and resource consumption (FPGA) [115]; Double precision performance (FPGA) [76]
Stencil computations	Cluster implementation (GPU) [11]; Varying stencil weights (CBEA, GPU) [43]; Domain blocking, register cycling (GPU) [118]; Domain blocking, loop unrolling (CBEA) [12]; Auto-tuning (CBEA, GPU) [46]; Time skewing (CBEA) [45,96]
Random numbers	Sub-word operations (FPGA) [169]; Generating initial state [111]; Stateless (GPU) [165]; Tausworthe (GPU) [85]; Architectural constraints (GPU, CPU, FPGA) [168]
Scan	Up and down sweep (GPU) [67,156]; Linked lists (CBEA) [19]; Histogram pyramids (GPU) [50,186]
Sorting	Bitonic sort (GPU) [147]; Hybrid radix-bitonic, out-of-core (GPU) [65]; Radix (GPU) [154]; AA-sort (CBEA) [93]
Image processing	Canny edge detection (GPU, FPGA) [109,123]; OpenCV (GPU, CBEA) [2,57,163]; Computer Vision library (GPU) [16]

Note: The table is not an exhaustive list, but gives an overview of state-of-the-art problems and techniques.

graphics memory and shared memory, but also to reduce movement from shared memory to registers. Contrary to general guidelines, their approach consumes a considerable amount of registers, reducing occupancy (see Section 3.1), yet provides unprecedented performance. A subset of BLAS has also been implemented on the CBEA. Chen et al. [41], Hackenberg [70] and Alvaro et al. [5] present highly optimized matrix multiplication implementations, where the latter uses 99.8% of the floating-point capacity of the SPEs for $C = C - A \times B$. On FPGAs, both fixed and floating point matrix multiplication has been proposed. Zhuo and Prasanna [185] present blocked matrix multiplication. Using *processing elements* consisting of one floating-point multiplier, one floating point adder and a set of registers, they optimize performance with respect to clock speed and memory bandwidth use. The use of more processing elements negatively affects attainable clock speed, whilst increased block size positively affects memory bandwidth use. Their design is further extendible to use several FPGAs, each with a separate local memory, and they attain 4 gigaflops performance on a Virtex-II XC2VP100.

Matrix multiplication is also a key element in the Linpack benchmark, used to classify computers on the Top500 list. Kistler et al. [99] implemented the benchmark on the CBEA-based Roadrunner supercomputer [21]. Using a matrix multiplication algorithm similar to that of Alvaro et al. [5], they achieved 75% of theoretical peak performance for the full benchmark.

LAPACK is a commonly used library for high-level algorithms from numerical linear algebra, that relies on parallel BLAS implementations to benefit from multi-core and heterogeneous architectures. To take full advantage of parallel BLAS implementations, however, LAPACK must be adapted to avoid unnecessary memory allocations and copying of data. PLASMA [141] is a new initiative to develop a linear algebra package that is more suitable for multi-core and heterogeneous architectures. Buttari et al. [34] describe how operations like Cholesky, LU and QR factorizations can be broken into sequences of smaller tasks, preferably Level 3 BLAS operations. A dependency graph is used to schedule the execution of tasks, so individual tasks can be assigned to one or more accelerator cores. On the FPGA, LU decomposition has been implemented by Zhuo and Prasanna [185]. They use a linear array of processor elements consisting of a floating point multiplier and a floating point adder. The first processor element also includes a floating point divider, requiring large amounts of real-estate. The whole design is limited by the frequency of the floating point divider for a small number of processing elements. A problem with using many processing elements is that the complex wiring degrades clock frequency. Going from one to 18 processing elements, the clock frequency is degraded by 25%. A mixed precision LU decomposition was developed by Sun et al. [164], where the FPGA calculates a low-precision approximation to the solution using the Dolittle algorithm, followed by an iter-

ative refinement step on the CPU. Using lower precision for most calculations on the FPGA lessens latencies of floating point units and uses less bandwidth, thus speeding up computations.

Whilst LAPACK is a highly efficient library, it requires low-level programming. MATLAB addresses this by offering a high-level interactive interface. There have been several efforts to accelerate MATLAB using GPUs, where native MATLAB syntax is used to transparently execute computations on the GPU. The first approach was given by Brodtkorb [32], providing a MATLAB extension calling an OpenGL-based backend to carry out the computations. A similar approach is taken in Jacket [1], a commercial product from Accelereyes.

Sparse linear algebra is most often bandwidth limited, as random sparse matrices are typically stored using a format such as compressed row storage (CRS). The CRS format consists of one vector with the values of all non-zero entries, stored row wise, and a second vector containing the column index of each entry. The start index of each row is stored in a third vector, yielding a very compact storage format. However, such storage formats require three lookups to find the value of an element: first the row, then the column, and finally the value. Due to such indirections, it is common with less than 10% bandwidth utilization, and less than 1% compute utilization when computing with sparse matrices on the CPU [64]. An important algorithm in sparse linear algebra is matrix-vector multiply, often used in matrix solvers such as the conjugate gradients algorithm. Williams et al. [182] explore a range of blocking strategies to avoid cache misses and improve performance of sparse matrix-vector multiply on multi-core platforms. They use asynchronous DMA transfers on the CBEA to stream blocks of matrix elements in and blocks of vector elements out of each SPE. A branchless technique similar to that of Blelloch et al. [27] is used to avoid mis-predictions, and SIMD instructions are used to further increase performance. They are, on average, able to utilize 92.4% of the peak bandwidth for a wide range of sparse matrices. This is far greater than the bandwidth utilization of multi-core CPUs when similar techniques are employed. On the GPU, Bell and Garland [23] explore sparse matrix-vector product using CUDA. They investigate a wide range of data structures including variants of CRS. However, the variants of CRS suffers either from non-coalesced memory reads or unbalanced workloads within warps. To overcome this, they developed a hybrid format, where structured parts of the matrix are stored using a dense format, and the remaining non-zero elements in a CRS format. Using this technique, a single NVIDIA GTX 280 GPU was twice as fast as two CBEAs running the aforementioned algorithm of Williams et al. [182].

5.2. The fast Fourier transform

The fast Fourier transform (FFT) is at the heart of many computational algorithms, such as solving differential equations in a Fourier basis, digital signal processing, and image processing. The FFT is usually among the first of algorithms ported to a new hardware platform, and efficient vendor-specific libraries are widely available.

NVIDIA provides the CUFFT library [127] for CUDA-enabled GPUs, modeled on the FFTW library [56]. Version 2.3 supports single and double precision complex values, while real values are supported for convenience only, as the conjugate symmetry property is not used. Nukada and Matsuoka [124] present an auto-tuning 3D FFT implementation that is 2.6–8 times faster than CUFFT version 2.1, and with only small performance differences between power-of-two and non-power-of-two FFTs. Their implementation is 2.7–16 times faster on a GTX 280, compared to FFTW running on a 2.2 GHz AMD quad core Phenom 9500. Govindaraju et al. [66] have implemented three different FFT approaches using CUDA; shared memory, global memory, and a hierarchically decomposed FFT. At runtime, they select the optimal implementation for the particular FFT. They report surpassing CUFFT 1.1 on both performance and accuracy. On a GTX280, they report 8–40 times speedup over a 3.0 GHz Intel Core2 Quad Extreme using Intel MKL. Volkov and Kazian [181] employ the approach taken by Volkov and Demmel [180], where register use in CUDA is increased, and report results equivalent to that of Govindaraju et al. [66]. Lloyd, Boyd and Govindaraju [107] have implemented the FFT using DirectX, which runs on GPUs from different vendors, and report speeds matching CUFFT 1.1.

IBM has released FFT libraries for the CBEA that can handle both 1D, 2D and 3D transforms [90,91]. The FFTW library includes an implementation for the CBEA [56]. Bader and Agarwal [18] present an iterative out-of-place FFT implementation that yields 18.6 gigaflops performance, several gigaflops faster than FFTW, and claim to be the fastest FFT implementation for 1–16,000 complex input samples. Xu et al. [184] have another approach where they address

communication latencies. By using indirect swap networks, they are able to halve communication overheads.

Both Altera and Xilinx provide FFT IP cores [3,4] that can be set up for both single precision and fixed point FFT computations of vectors of up-to 65,536 elements. McKeown and McAllister [115] optimize FFT performance with respect to power and resource consumption, and show over 36% reduction in power consumption, and 61% reduction in slice use compared to the 64-point Xilinx IP core. Hemmert and Underwood [76] have given some remarks on double precision FFTs on FPGAs, discussing pipelined versus parallel designs. They report that the optimal design depends on both the FFT size, as well as the FPGA fabric size.

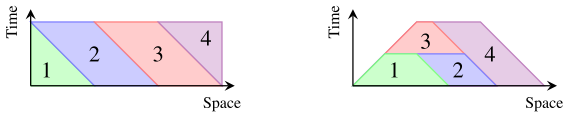
5.3. Stencil computations

Stencil computations are central in many computational algorithms, including linear algebra, solving partial differential equations, and image processing algorithms. Basically, a stencil computation is a weighted average of a neighbourhood, computed for every location in a grid. The stencil determines the size and weighing of the neighbourhood, and all grid points can be computed in parallel by using a separate input and output buffer. The computational complexity of stencil computations is often low, making them memory bound, and thus, leveraging the high bandwidth of accelerators can yield performance gains. As an extreme example, Mattson et al. [112] demonstrated one teraflops performance whilst using a mere 97 watts of power on the Intel Teraflops Research Chip [176].

Stencil computations can be used efficiently to solve partial differential equations that are discretized using an explicit finite difference approach. Hagen et al. [71, 72] explored the suitability of early GPUs for complex finite difference schemes to solve the shallow water and Euler equations. Speed-ups of 10–30 times compared to equivalent CPU implementations were presented, even on non-trivial domains. The Asian Disaster Preparedness Center and the Japan Meteorological Agency solve the shallow water equations in their prototype tsunami simulator [11]. Their single node GPU implementation is 62 times faster than the corresponding CPU version, and they estimate that a simulation of a 6000×6000 mesh will run in less than 10 min using a cluster of 32 GPUs. Christen et al. [43] explore stencil computations in a bio-medical simulation where the stencil weights vary in space. This makes the computation far more complicated compared to a static stencil, yet they report 22 gigaflops performance using two CBEAs on a QS22 blade, and 7.8 gigaflops on the NVIDIA GeForce 8800 GTX. For comparison, a dual quad-core Intel Clovertown achieves 2 gigaflops for the same computation.

Araya-Polo et al. [12] and Micikevicius [118] solve the linear wave equation in three dimensions on the CBEA and GPU, respectively. The wave equation is key to reverse time migration in seismic processing, and they solve it using an explicit finite difference scheme that ends up in a stencil computation. On the CBEA, Araya-Polo et al. divide the computational domain into blocks, where the SPEs independently compute on separate blocks. They traverse each block in 2D planes, and are able to hide memory transfers by explicitly overlapping communication with computations. They also employ extensive loop unrolling enabled by the large number of registers, and report 21.6 GB/s and 58.3 gigaflops performance, corresponding to 85% and 25% of peak performance for bandwidth and arithmetics, respectively. Their implementation runs eight times faster than an equally tuned CPU counterpart on a 2.3 GHz PowerPC 970MP. On the GPU, Micikevicius describes another approach, where the volume is similarly divided into blocks. Within each block, the volume is traversed in a single pass by cycling over 2D slices. Shared memory is used to store the 2D slice, including the apron as defined by the stencil. The depth dimension is stored in per-thread registers, which are used as a cyclic buffer. Consistent results of 45–55 GB/s, and up-to 98 gigaflops floating point performance are reported. This corresponds to around 50% and 9% of peak bandwidth and arithmetic performance, respectively, and the bottleneck appears to be global memory latency. A GPU cluster implementation is also presented. Super-linear speedup over one GPU is achieved as each card uses less memory, thus causing fewer translation lookaside buffer (TLB) misses.

Datta et al. [46] employ an auto-tuning approach to optimize performance of a seven-point 3D stencil computation on multi-core CPUs, the CBEA, and a GPU. In double precision they achieve 15.6 gigaflops on a QS22 blade, and 36.5 gigaflops on the NVIDIA GTX 280 GPU. However, whilst they report that the CBEA is memory bound, they conclude that the GPU is computationally bound for double precision. In total the GPU is 6.8 times faster than their best auto-tuned code on a traditional symmetric multi-core platform, a dual socket Sun UltraSparc T2+ solution with a total of 16 cores.



(a) Time-skewing (b) Cache Oblivious

Fig. 6. Cache-aware and cache-oblivious blocking algorithms.

Traditional cache based architectures can employ algorithmic optimizations such as time skewing [116] to increase cache reuse. Time skewing requires the user to explicitly partition the domain into space–time parallelograms, as shown in Fig. 6(a). After partitioning, the stencil kernel is executed on each parallelogram in turn. This concept has been further developed into recursive cache-oblivious algorithms that subdivide the trapezoids until they contain only one time-step [60], as illustrated in Fig. 6(b). Both cases can be extended to parallel execution by processor overlap. Time skewing was mapped to the CBEA by Kamil et al. [96] and Datta et al. [45]. In their early work, they used four step time skewing with overlapping parallelograms between SPEs to solve a three-dimensional heat diffusion problem using a finite difference scheme. They demonstrate 65.8 gigaflops in single precision for the 3.2 GHz CBEA. For double precision calculations, the early CBEA chips were computationally bound, and thus, do not benefit from memory optimizations.

5.4. Random number generation

Monte Carlo integration methods are based on function evaluations at randomly sampled points. The idea is to run the same simulation numerous times continuously drawing new random numbers. In principle, this approach is embarrassingly parallel. However, the production of random numbers, which fuels the process, is usually not embarrassingly parallel.

Pseudorandom number generators approximate truly random numbers, and are often formulated as forms of recurrence relations, maintaining a state of the m most recently produced numbers and tapping into this state to produce a new pseudorandom number. Linear congruential generators are defined by the simple recurrence relation $x_i = (ax_{i-1} + c) \bmod m$, producing sequences with a modest maximum period of m. Multiple recursive generators combine the output of multiple generators, extending the overall period to the smallest common multiplier of the periods of the combined generators. The Mersenne Twister [110] is based on a linear recurrence function tapping into a state of 624 32-bit integers. Its output is of high quality, and with a very long period, it is often the generator of choice.

Manipulation at word level is the natural primitive for most processors, and thus, most pseudorandom number generators work on the level of words as well. A variant of the Mersenne Twister is the SIMD-oriented Fast Mersenne Twister [153], which utilizes the 128-bit operations provided by SSE2 and AltiVec, and roughly doubles the performance. On the other hand, FPGAs provide very fine-grained binary linear operations, which opens possibilities for sub-word operations, investigated by, e.g., Thomas and Luk [169].

Recurrence relations are serial in nature, but a common strategy is to run a set of pseudorandom number generators in parallel, taking care when initializing the generators to avoid correlations. For the Mersenne Twister, Matsumoto and Nishimura [111] proposed a scheme to generate the initial state for an array of independent generators. Further, the size of the generator's state may pose a problem: On NVIDIA GPUs, a single warp of 32 threads, each running the Mersenne Twister, would require 78 KiB of memory just for the state, almost five times the amount of shared memory.

The output of pseudorandom number generators is usually uniformly distributed. There are various approaches to convert uniformly distributed numbers to other distributions. One example is a rejection method, which rejects samples that do not fit the distribution. This could introduce code path divergences on GPUs: The Ziggurat method has only 1% risk of divergent behaviour, but with 32 threads in a warp, the risk of a divergent warp is 30% [168]. On FPGAs, such a varying amount of outputs per invocation requires the implementation to handle pipeline bubbles. A more direct approach is to use the inverse cumulative distribution function. For the normal distribution, a closed expression for the inverse cumulative distribution function does not exist, so one must approximate using piecewise rational polynomials. The Box–Muller transform uses trigonometric operations and logarithms, which are generally considered expensive. On GPUs, however, fast versions of these instructions are built in, as they are very common in computer graphics.

Most random number generators can be used directly on the CBEA, and the CBEA SDK [89] contains a Monte Carlo library with implementations of the Mersenne Twister, the SIMD-oriented Fast Mersenne Twister, Kirkpatrick–Stoll pseudorandom number generators, as well as the Box–Muller transform and transformation using the inverse cumulative normal distri-

bution. On GPUs, the early approach of Sussman et al. [165] to implementing random number generation used stateless combined explicit inverse congruential generator. Howes and Thomas [85] give an overview of implementing pseudorandom number generators in CUDA, pointing to the problems of large state vectors and that integer modulo is expensive on current GPUs, and propose a hybrid generator that combines a Tausworthe generator [106] with linear congruential generators. The CUDA SDK [128] contains an implementation running an array of Mersenne Twisters in parallel, storing the states in local memory. The Brook+ SDK [8] contains an implementation of the SIMD-oriented Fast Mersenne Twister, suitable for the superscalar architecture of the AMD GPUs. Further, Thomas et al. [168] evaluate strategies for generating pseudorandom numbers on CPUs, GPUs, FPGAs and massively parallel processor arrays, focusing on algorithmic constraints imposed by the respective hardware architectures. They give an interesting benchmark, comparing both raw performance and energy efficiency for their chosen approaches. An Intel Core 2 CPU produces on average 1.4 gigasamples/s at an energy efficiency of 5 megasamples/joule, an NVIDIA GTX 280 produces 14 gigasamples/s at 115 megasamples/joule, and a Xilinx Virtex-5 FPGA produces 44 gigasamples/s at 1461 megasamples/joule.

5.5. *Data-parallel primitives and sorting*

Data-parallel primitives are building blocks for algorithm design that execute efficiently on data-parallel architectures. Particularly common patterns are reduce, partial sums, partition, compact, expand and sorting.

The reduce pattern reduces an array of values into a single result, e.g., calculating the sum of an array of numbers. In this case, the direct approach of adding numbers one-by-one, $(((a_0 + a_1) + a_2) + \cdots + a_{n-1})$ has both work and step efficiency of $\mathcal{O}(n)$. However, if we do pairwise recursive additions instead, shown in Fig. 7, the step efficiency is reduced to $\mathcal{O}(\log_2 n)$ if we can carry out n additions in parallel. Pairwise recursive addition is an example of the binary reduce pattern, which works with any binary associate operator.

Prefix-sum scan, initially proposed as part of the APL programming language [95], computes all partial sums, $[a_0, (a_0 + a_1), (a_0 + a_1 + a_2), \ldots, (a_0 + \cdots + a_{n-1})]$, for any binary associative operator. The approach of Hillis and Steele [81] computes scan in $\mathcal{O}(\log_2 n)$ steps using $\mathcal{O}(n \log_2 n)$ arithmetic operations. This approach was implemented by Horn [84] in

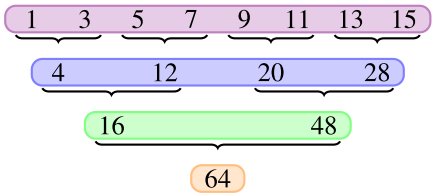


Fig. 7. Reduction to sum n elements requires $\log(n)/\log(p)$ passes, where p is the reduction factor per pass.

OpenGL, producing the first efficient implementation of scan on the GPU. Another implementation is provided in the Brook+ SDK [8]. Blelloch [26] presents a two-phase approach of scan using an up-sweep phase and a down-sweep phase. With this approach, the number of arithmetic operations is reduced to $\mathcal{O}(n)$. Sengupta et al. [157] proposed an OpenGL implementation of this approach, and Greß et al. [67] proposed another variant based on 2D-arrays and 4-way reductions, also using OpenGL. Harris et al. [75] proposed an implementation optimized for CUDA, which has been incorporated in the CUDA Data Parallel Primitives Library [74]. Blelloch [26] presents a range of uses for scan, complemented by Sengupta et al. [156] with particular emphasis on GPU implementation. On the CBEA, Bader et al. [19] investigate prefix-sum scan of arbitrary linked lists. They partition the lists into a set of sublists, and find the local prefix-sum of each sublist in parallel, using software-managed threads (see Section 4.3). In a subsequent step, the prefix-sum of the sums of the sublists are computed, which is then used to update local prefix-sums forming the global prefix-sum.

The partition pattern can be implemented using scan. For example, Blelloch [26] suggests implementing the split phase of radix sort using scan. In step i of radix sort, the data is partitioned into two sets depending on whether bit i is set or not. By creating an array containing 1 for every element with bit i not set and 0 otherwise, the $+$-scan of this array produces the offsets in the resulting partitioned array for the 0-bit elements. The $+$-scan is simply scan using the addition operator. Running a $+$-scan in reverse order produces the offsets from the end of the output array for the 1-bit elements, and these two scans can be carried out in parallel.

Compaction is a variation of partition in which the output is a compact array containing only a subset of the elements in the input array, and expansion is the situation where an input stream element is expanded to multiple elements in the output stream. Both can be done using scan with a predicate that associates input

stream elements with the output stream element count for that element. For compaction, this amounts to associating 1 with elements that should be kept and 0 with the elements to be discarded. Running scan on the result of the predicate produces the offset into the compacted array for each input array element to be kept, and by iterating over the input elements, consulting the results of the predicate function and scan, elements to be kept can be written to its location in the compacted array. This operation requires writing to arbitrary memory locations, that is, scatter writes, which was unavailable on early GPUs. Horn [84] circumvented this by iterating over the compacted elements, using binary search on the scan output to find the corresponding input array element. Ziegler et al. [186] proposed a method for this problem in OpenGL using histogram pyramids. Histogram pyramids use 4-way reduction to produce a pyramid of partials sums, similar to the up-sweep phase of Greß et al. [67], however, the down-sweep phase is omitted. The algorithm iterates over the compacted array and the position of the corresponding input element is found by traversing the histogram pyramid from top to bottom. An extension to the OpenGL histogram pyramid algorithm allowing stream expansion was proposed by Dyken et al. [50], along with a variant adapted to the CUDA architecture.

Another important algorithmic primitive is sorting a one-dimensional vector, a common task in a wide range of applications. However, traditional sorting algorithms developed for the CPU do not map directly to new architectures. Early GPGPU sorting algorithms implemented using graphics APIs could not output data to arbitrary memory locations. Instead, Purcell et al. [147] implemented bitonic sort, where the basic operation is to compare two values and output the smallest or largest element. Govindaraju et al. [65] implemented a hybrid radix-bitonic sort to sort out-of-core data sets, allowing the sorting key to be larger than 32-bit. In their implementation, the GPU mainly performs a bitonic sort.

The introduction of shared memory opened up the possibility to implement a wider range of algorithms directly on GPUs. Generally, the algorithms are based on first sorting a block of data that fits in shared memory, and then merge blocks. This strategy is also suited for other parallel architectures, such as multi-core CPUs and the CBEA, and is similar to algorithms used for traditional distributed memory systems. On the GPU, Satish et al. [154] base their implementation on radix sort, and use the aforementioned data parallel primitives in what they claim is the fastest sorting implementation written in CUDA. AA-sort by Inoue et al. [93] target CBEA and multi-core processors with SIMD capabilities. In their implementation, data blocks fitting in cache/local store are sorted using a SIMD version of comb sort. Chhugani et al. [42] presented a sorting algorithm suitable for parallel architectures with a shared and coherent L2 cache and wide SIMD capabilities. The upcoming Larrabee GPU is an example of such an architecture. First, they divide the data into blocks that fit in L2 cache and sort each block. Each block is split once more, to allow each core to work on one sub-block. Then the cores cooperate on merging the sorted sub-blocks. This structure allows the blocks to reside in L2 cache until the entire block is sorted.

5.6. Image processing and computer vision

Many algorithms within image processing and computer vision are ideal candidates for heterogeneous computing. As the computational domain often is regular and two-dimensional, the algorithms often include embarrassingly parallel stages, in addition to high-level reasoning. One example is Canny edge detection [35], which itself is often used as input to other algorithms. In Canny edge detection, a blurred gradient image is created by convolving the input with a Gauss filter, followed by a gradient convolution filter. Local maxima in the gradient direction are considered candidate edges. To distinguish between false and true edges, a two-step hysteresis thresholding called non-maximum suppression is performed. Magnitudes larger than the upper threshold are considered true edges, and magnitudes lower than the lower threshold are considered non-edges. The remaining edges are then classified as edges only if they can be directly connected to existing true edges. This is an iterative process, where the real edges grow out along candidate edges.

Luo and Duraiswami [109] present a full implementation of Canny edge detection on the GPU. The convolution steps are fairly straightforward, and are efficiently implemented using techniques described in the NVIDIA CUDA Programming Guide. However, they operate on RGB images with eight bits per channel, whereas the memory bus on the GPU is optimized for 32-bit coalesced reads. They solve this issue by using 25% fewer threads during memory read, but where each thread reads 32 bits. These 32 bits are subsequently split into the respective eight bit color channels using efficient bit shifts. Thus, they require no redun-

dant memory reads, and the access is coalesced. For the final part of the algorithm, they use a four-pass algorithm to connect edges across CUDA block boundaries. Within each block, the threads cooperate using a breadth first search, and edges that cross boundaries are connected between the passes. On an NVIDIA 8800 GTX, the implementation executed over three times faster compared to an optimized OpenCV implementation on an 2.4 GHz Intel Core 2 Duo for different test images. The bottleneck is the edge connection step, which is a function of the number of edges, and dominates 70% of the computational time. On the FPGA, Neoh and Hazanchuk [123] utilize a pipelining approach to find edges. Both convolutions are implemented using symmetric separable filters in a pipeline where latency is traded for throughput. Connecting candidate edges is done using a FIFO queue initially containing all classified edges. For each element in the FIFO queue, candidate edges in the three by three neighbourhood are found, marked as true edges and added to the FIFO queue. On the Altera Stratix II FPGA, the implementation is capable of processing 4000 256×256 images per second.

Canny edge detection is, as mentioned, often an input to other algorithms for high-level reasoning, commonly referred to as computer vision. OpenCV [28] is an open source computer vision library, focusing on real-time performance with a highly tuned CPU implementation. OpenCV has also been mapped to both GPUs and the CBEA. GpuCV [2] uses GLSL, CUDA, or a native OpenCV implementation, and can automatically switch to the best implementation at run-time with a small overhead. Operators such as Erode and Sobel are benchmarked, and show speedups of 44–193 times on an NVIDIA GTX 280 compared to a 2.13 GHz Core2 Duo. CVCell is a project that consists of CBEA accelerated plugins for OpenCV [57]. Speed-ups of 8.5–37.4 times compared to OpenCV on a 2.66 GHz Intel Core 2 Duo are reported for morphological operations [163]. Finally, MinGPU [16], is a computer vision library built on top of OpenGL. This preserves portability, but often sacrifices performance, as OpenGL initiation can be more costly compared to CUDA.

It becomes increasingly important to design image processing algorithms for heterogeneous architectures, for example, using total variation methods instead of graph cut methods [145] for applications such as optical flow computation, image registration, 3D reconstruction and image segmentation. Michel et al. [117] use the GPU to enable a humanoid robot to track 3D objects. Color conversion, Canny edge detection, and pose estimation is run on the GPU, and iterative model fitting is run on the CPU. This enables real-time performance, and leaves enough CPU resources to calculate footsteps and other tasks. Another example is Boyer et al. [31], who use the GPU to speed up automatic tracking of white blood cells, important in research into inflammation treatment drugs. Currently, this is a manual process, where the center of each white blood cell is marked manually. They employ a Gradient Inverse Coefficient of Variation algorithm to locate the cells initially, and in subsequent frames track the boundary. They isolate a rectangle around the original cell position, and compute a snake algorithm on rectangles in parallel. They report 9.4 and 27.5 times speedup on an NVIDIA GTX 280 compared to an OpenMP implementation running on a 3.2 GHz Intel Core 2 Quad Extreme. Finally, Baker et al. [20] compared FPGAs, GPUs, and the CBEA for matched filter computations, used to analyze hyperspectral images. The algorithm is partitioned onto the CPU and accelerator core, and they report the NVIDIA 7900 GTX to be most cost-efficient, and the CBEA to give the best speedup and speedup per watt.

6. Summary

We have described hardware, software, and state-of-the-art algorithms for GPUs, FPGAs and the CBEA in the previous sections. In this section, we summarize the architectures, give our view on future trends, and offer our concluding remarks.

6.1. Architectures

The three architectures we have reviewed have their distinct benefits and drawbacks, which in turn affect how well they perform for different applications. The major challenge for application developers is to bridge gap between theoretical and experienced performance, that is, writing well balanced applications where there is no apparent single bottleneck. This, of course, varies from application to application, but also for a single application on different architectures. However, it is not only application developers that have demands to architectures: economic, power and space requirements in data-centers impose hard limits to architecture specifications.

The GPU is the best performing architecture when it comes to single precision floating point arithmetic,

with an order of magnitude better performance compared to the others. When considering the price and performance per watt, the GPU also comes out favourably. However, the GPU performs best when the same operation is independently executed on a large set of data, disfavouring algorithms that require extensive synchronization and communication. GPU double precision performance is now available, with one fifth of the single precision performance for AMD, and one half for NVIDIA. The new programming environments, such as CUDA and Brook+, enable efficient development of code, and OpenCL further offers hope for code portability.

The CBEA is still twice as fast as the state-of-the-art CPU, even though it is now three years old. Furthermore, it offers a very flexible architecture where each core can run a separate program. Synchronization and communication is fast, with over 200 GB/s bandwidth on the element interconnect bus. This makes the CBEA extremely versatile, yet somewhat more difficult to program than the GPU. It is also a very well performing architecture when it comes to double precision. However, it is rather expensive.

FPGA performance is difficult to quantify in terms of floating point operations, as floating point is typically avoided. FPGAs are much more suited for algorithms where fixed point, integer, or bit operations are key. For such tasks, the FPGA has an outstanding raw performance, and an especially good performance per watt ratio. If one needs floating point operations, FPGA implementations benefit from tuning the precision to the minimum required level, independent of the IEEE-754 standard. However, FPGAs have a discouraging price tag. They can also be very difficult to program, but new languages such as Mitrion-C and Viva offer promising abstractions.

6.2. Emerging features and technology

We believe that future applications will require heterogeneous processing. However, one must not overlook legacy software that has existed for decades. Such software is economically unfeasible to redesign, and must use library calls to benefit from heterogeneous processing. This has the drawback captured in Amdahl's law, where the serial part of the code quickly becomes the bottleneck. New applications and algorithms, however, can be designed for existing, and future, architectures.

In the future, we see it as likely that GPUs will enter the HyperTransport bus, similarly to current FPGAs. We believe that such GPU cards on the HyperTransport bus will contain separate high-speed memory, and target server markets at a higher premium. Using NUMA technology, we even see the possibility of a shared address space, where both the CPU and GPU can transparently access all memory. Such an architecture would indeed be extremely useful, alleviating the PCI express bottleneck, but more importantly the issue of distinct memory spaces. Furthermore, it will be interesting to see the impact of the Larrabee on the future of GPU computing.

The CBEA roadmap schedules a chip with two Power processor cores and 32 synergistic processing elements for 2010. The state of these plans is uncertain, but we find it likely that similar designs will be used in the future. The major strengths of the CBEA is the versatility of the synergistic processing units, and the use of local store memory in conjunction with DMA. We believe such features will become increasingly used as power constraints become more and more pressing.

The obvious path of FPGAs is to simply continue increasing the clock frequency, and decrease the production techniques. However, the inclusion of power processors in hardware is an interesting trend. We believe that the trend of an increased number of special-purpose on-chip hardware, such as more floating point adders, will continue. This will rapidly broaden the spectrum of algorithms that are suitable for FPGAs.

6.3. Concluding remarks

Heterogeneous computing has in very few years emerged as a separate scientific field encompassing existing fields such as GPGPU. One of the reasons that heterogeneous computing has succeeded so well until now has been the great success of GPGPU. However, the field of GPGPU is focused only on the use of graphics cards. In the future, we believe that algorithms cannot be designed for graphics cards alone, but for general heterogeneous systems with complex memory hierarchies. We do not believe that symmetric multiprocessing has a future in the long term, as it is difficult to envision efficient use of hundreds of traditional CPU cores. The use of hundreds of accelerator cores in conjunction with a handful of traditional CPU cores, on the other hand, appears to be a sustainable roadmap.

We have little belief in the one-size-fits-all for scientific computing. The three architectures we have described are currently addressing different needs, which we do not see as transient. The GPU maximises highly parallel stream computing performance, where com-

munication and synchronization is avoided. The CBEA offers a highly versatile architecture, where each core can run a separate program with fast inter-core communication. Finally, the FPGA offers extreme performance for applications relying on bit, integer, logic and lower precision operations.

Acknowledgements

The authors would like to thank Gernot Ziegler at NVIDIA Corporation, Knut-Andreas Lie and Johan Seland at SINTEF ICT, and Praveen Bhaniramka and Gaurav Garg at Visualization Experts Limited for their fruitful comments and feedback. We also appreciate the valuable input from the anonymous reviewers, and the continued support from AMD, IBM and NVIDIA. Part of this work is done under Research Council of Norway project number 180023 (Parallel3D) and 186947 (Heterogeneous Computing). Dr. Storaasli's research contributions were sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory managed by UT-Battelle for the US Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Accelereyes, Jacket user guide, February 2009.

[2] Y. Allusse, P. Horain, A. Agarwal and C. Saipriyadarshan, GpuCV: A GPU-accelerated framework for image processing and computer vision, in: *Intl. Symp. on Advances in Visual Computing*, Springer-Verlag, Berlin, 2008, pp. 430–439.

[3] Altera, FFT megacore function user guide, March 2009.

[4] Altera, Logicore ip fast Fourier transform v7.0 user guide, June 2009.

[5] W. Alvaro, J. Kurzak and J. Dongarra, Fast and small short vector SIMD matrix multiplication kernels for the synergistic processing element of the cell processor, in: *Intl. Conf. on Computational Science*, Springer-Verlag, Berlin, 2008, pp. 935–944.

[6] AMD, R700-family instruction set architecture, March 2009.

[7] AMD, ATI Radeon HD 5870 GPU feature summary, available at: http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx (visited 2009-10-28).

[8] AMD, ATI stream software development kit, available at: http://developer.amd.com/gpu/ATIStreamSDK/ (visited 2009-04-28).

[9] AMD, Stream KernelAnalyzer, available at: http://developer.amd.com/gpu/ska/.

[10] AMD, AMD core math library for graphic processors, March 2009, available at: http://developer.amd.com/gpu/acmlgpu/ (visited 2009-04-20).

[11] T. Aoki, Real-time tsunami simulation on a multinode GPU cluster, in: *SuperComputing*, Portland, OR, 2009, poster.

[12] M. Araya-Polo, F. Rubio, R. de la Cruz, M. Hanzich, J. Cela and D. Scarpazza, 3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors, *Sci. Prog.* **17**(1,2) (2009), 185–198.

[13] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams and K. Yelick, The landscape of parallel computing research: A view from Berkeley, Technical report, EECS Department, University of California, Berkeley, December 2006.

[14] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, E. Lee, N. Morgan, G. Necula, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel and K. Yelick, The parallel computing laboratory at U.C. Berkeley: A research agenda based on the Berkeley view, Technical report, EECS Department, University of California, Berkeley, December 2008.

[15] W. Aspray, The Intel 4004 microprocessor: What constituted invention?, *Hist. Comput.* **19**(3) (1997), 4–15.

[16] P. Babenko and M. Shah, MinGPU: a minimum GPU library for computer vision, *Real-Time Image Process.* **3**(4) (2008), 255–268.

[17] J. Backus, Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs, *Commun. ACM* **21**(8) (1978), 613–641.

[18] D. Bader and V. Argwal, FFTC: Fastest Fourier transform for the IBM cell broadband engine, in: *Intl. Conf. on High Performance Computing*, Goa, India, 2007, pp. 172–184.

[19] D. Bader, V. Agarwal and K. Madduri, On the design and analysis of irregular algorithms on the cell processor: A case study of list ranking, in: *Intl. Parallel and Distributed Processing Symp.*, Long Beach, CA, USA, 2007, pp. 1–10.

[20] Z. Baker, M. Gokhale and J. Tripp, Matched filter computation on FPGA, cell and GPU, in: *Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society, Washington, DC, USA, 2007, pp. 207–218.

[21] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin and J. Sancho, Entering the petaflop era: The architecture and performance of Roadrunner, in: *Supercomputing*, November 2008, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–11.

[22] J. Beeckler and W. Gross, Particle graphics on reconfigurable hardware, *Reconfigurable Technology and Systems* **1**(3) (2008), 1–27.

[23] N. Bell and M. Garland, Efficient sparse matrix-vector multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[24] P. Bellens, J. Perez, R. Badia and J. Labarta, CellSs: a programming model for the cell BE architecture, in: *Supercomputing*, ACM, New York, NY, USA, 2006, p. 86.

[25] S. Benkner, E. Laure and H. Zima, HPF+: An extension of HPF for advanced applications, Technical report, The HPF+ Consortium, 1999.

[26] G. Blelloch, Prefix sums and their applications, Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[27] G. Blelloch, M. Heroux and M. Zagha, Segmented operations for sparse matrix computation on vector multiprocessors, Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1993.

[28] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, O'Reilly, Cambridge, MA, USA, 2008.
[29] F. Bodin, An evolutionary path for high-performance heterogeneous multicore programming, 2008.
[30] G. Boone, Computing systems CPU, United States Patent 3,757,306, August 1971.
[31] M. Boyer, D. Tarjan, S. Acton and K. Skadron, Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors, in: *Int. Parallel and Distributed Processing Symp.*, Rome, Italy, 2009, pp. 1–12.
[32] A. Brodtkorb, The graphics processor as a mathematical coprocessor in MATLAB, in: *Intl. Conf. on Complex, Intelligent and Software Intensive Systems*, Barcelona, Spain, IEEE Computer Society, 2008, pp. 822–827.
[33] I. Buck, T. Foley, D. Horn, J. Sugerman, M. Houston and P. Hanrahan, Brook for GPUs: Stream computing on graphics hardware, SIGGRAPH, Los Angeles, CA, 2004.
[34] A. Buttari, J. Langou, J. Kurzak and J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, *Parallel Comput.* **35**(1) (2009), 38–53.
[35] J. Canny, A computational approach to edge detection, *Pattern Anal. Machine Intelligence* **8**(6) (1986), 679–698.
[36] Celoxica website, http://www.celoxica.com/ (visited 2009-04-28).
[37] R. Chamberlain, M. Franklin, E. Tyson, J. Buhler, S. Gayen, P. Crowley and J. Buckley, Application development on hybrid systems, in: *SC'07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ACM, New York, NY, USA, 2007, pp. 1–10.
[38] Chapel language specification 0.782, Technical report, Cray Inc., 2009.
[39] B. Chapman, G. Jost and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, Cambridge, MA, USA, 2007.
[40] S. Che, J. Li, J. Sheaffer, K. Skadron and J. Lach, Accelerating compute-intensive applications with GPUs and FPGAs, in: *Symposium on Application Specific Processors, 2008 (SASP'2008)*, Anaheim, CA, June 2008, pp. 101–107.
[41] T. Chen, R. Raghavan, J. Dale and E. Iwata, Cell broadband engine architecture and its first implementation: a performance view, *IBM J. Res. Dev.* **51**(5) (2007), 559–572.
[42] J. Chhugani, A. Nguyen, V. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar and P. Dubey, Efficient implementation of sorting on multi-core SIMD CPU architecture, *Proc. VLDB Endowment* **1**(2) (2008), 1313–1324.
[43] M. Christen, O. Schenk, P. Messmer, E. Neufeld and H. Burkhart, Accelerating stencil-based computations by increased temporal locality on modern multi- and many-core architectures, in: *Intl. Workshop on New Frontiers in High-Performance and Hardware-Aware Computing*, KIT Scientific Publishing, Karlsruhe, Germany, 2008, pp. 47–54.
[44] Pico Computing, Accelerating bioinformatics searching and dot plotting using a scalable FPGA cluster, November 2009 (visited 2009-11-14).
[45] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf and K. Yelick, Optimization and performance modeling of stencil computations on modern microprocessors, *SIAM Rev.* **51**(1) (2009), 129–159.
[46] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf and K. Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: *Supercomputing*, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–12.
[47] J. Dongarra, Basic linear algebra subprograms technical forum standard, *High Perform. Appl. Supercomput.* **16** (2002), 1–111.
[48] U. Drepper, What every programmer should know about memory, November 2007, available at: http://people.redhat.com/drepper/cpumemory.pdf (visited 2009-03-20).
[49] Dsplogic website, http://www.dsplogic.com/ (visited 2009-04-28).
[50] C. Dyken, G. Ziegler, C. Theobalt and H.-P. Seidel, High-speed marching cubes using histogram pyramids, *Computer Graphics Forum* **27**(8) (2008), 2028–2039.
[51] EDA Industry Working Groups, http://www.vhdl.org/ (visited 2009-04-28).
[52] A. Eichenberger, J. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao and M. Gschwind, Optimizing compiler for the cell processor, in: *Intl. Conf. on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, Washington, DC, USA, 2005, pp. 161–172.
[53] A. Eichenberger, J. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, R. Archambault, Y. Gao and R. Koo, Using advanced compiler technology to exploit the performance of the cell broadband engine architecture, *IBM Syst. J.* **45**(1) (2006), 59–84.
[54] E. Elsen, P. LeGresley and E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, *Comput. Phys.* **227**(24) (2008), 10148–10161.
[55] K. Fatahalian, D. Horn, T. Knight, L. Leem, M. Houston, J. Park, M. Erez, M. Ren, A. Aiken, W. Dally and P. Hanrahan, Sequoia: programming the memory hierarchy, in: *Supercomputing*, ACM, New York, NY, USA, 2006, p. 83.
[56] Fftw website, http://www.fftw.org (visited 2009-04-28).
[57] Fixtars, OpenCV on the cell, available at: http://cell.fixstars.com/opencv/ (visited 2009-03-20).
[58] M. Flynn, Some computer organizations and their effectiveness, *Trans. Comput.* **C-21**(9) (1972), 948–960.
[59] M. Flynn, R. Dimond, O. Mencer and O. Pell, Finding speedup in parallel processors, in: *Intl. Symp. on Parallel and Distributed Computing*, Miami, FL, USA, July 2008, pp. 3–7.
[60] M. Frigo and V. Strumpen, The memory behavior of cache oblivious stencil computations, in: *Supercomputing*, Vol. 39, Kluwer Academic Publishers, Hingham, MA, USA, 2007, pp. 93–112.
[61] I. Foster and K. Chandy, Fortran M: A language for modular parallel programming, *Parallel Distrib. Comput.* **26** (1992).
[62] D. Göddeke, R. Strzodka and S. Turek, Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, *Parallel Emergent Distrib. Syst.* **22**(4) (2007), 221–256.
[63] D. Göddeke and R. Strzodka, Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations (part 2: Double precision GPUs), Technical report, Technical University Dortmund, 2008.
[64] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski and S. Turek, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, *Parallel Comput.* **33**(10,11) (2007), 685–699.

[65] N. Govindaraju, J. Gray, R. Kumar and D. Manocha, GPUTeraSort: high performance graphics co-processor sorting for large database management, in: *Intl. Conf. on Management of Data*, ACM, New York, NY, USA, 2006, pp. 325–336.

[66] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith and J. Manferdelli, High performance discrete Fourier transforms on graphics processors, in: *Supercomputing*, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–12.

[67] A. Greß, M. Guthe and R. Klein, GPU-based collision detection for deformable parameterized surfaces, *Computer Graphics Forum* **25**(3) (2006), 497–506.

[68] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd edn, MIT Press, Cambridge, MA, USA, 1999.

[69] J. Gustafson, Reconstruction of the Atanasoff–Berry computer, in: *The First Computers: History and Architectures*, R. Rojas and U. Hashagen, eds, MIT Press, Cambridge, MA, USA, 2000, pp. 91–106, Chapter 6.

[70] D. Hackenberg, Fast matrix multiplication on cell (SMP) systems, July 2007, available at: http://www.tu-dresden.de/zih/cell/matmul/ (visited 2009-02-24).

[71] T. Hagen, J. Hjelmervik, K.-A. Lie, J. Natvig and M. Henriksen, Visual simulation of shallow-water waves, *Simul. Model. Pract. Theory* **13**(8) (2005), 716–726.

[72] T. Hagen, K.-A. Lie and J. Natvig, Solving the euler equations on graphics processing units, in: *Intl. Conf. on Computational Science*, V.N. Alexandrov, G.D. van Albada, P.M. Sloot and J. Dongarra, eds, LNCS, Vol. 3994, Springer, 2006, pp. 220–227.

[73] M. Harris, Parallel computing with CUDA, SIGGRAPH Asia 2008 presentation, avaiklable at: http://sa08.idav.ucdavis.edu/NVIDIA.CUDA.Harris.pdf (visited 2009-04-28).

[74] M. Harris, J. Owens, S. Sengupta, Y. Zhang and A. Davidson, CUDPP: CUDA data parallel primitives library, available at: http://www.gpgpu.org/developer/cudpp/ (visited 2009-03-20).

[75] M. Harris, S. Sengupta and J. Owens, Parallel prefix sum (scan) with CUDA, in: *GPU Gems 3*, H. Nguyen, ed., Addison-Wesley, Boston, MA, USA, 2007, pp. 851–876.

[76] K. Hemmert and K. Underwood, An analysis of the double-precision floating-point FFT on FPGAs, in: *Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society, Washington, DC, USA, 2005, pp. 171–180.

[77] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 4th edn, Morgan Kaufmann, San Francisco, CA, USA, 2007.

[78] N. Higham, The accuracy of floating point summation, *Sci. Comp.* **14**(4) (1993), 783–799.

[79] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su and K. Yelick, Titanium language reference manual, Technical report, UC Berkeley, 2005.

[80] M. Hill and M. Marty, Amdahl's law in the multicore era, *IEEE Computer* **41**(7) (2008), 33–38.

[81] W. Hillis and G. Steele Jr., Data parallel algorithms, *Commun. ACM* **29**(12) (1986), 1170–1183.

[82] J. Hjelmervik, Heterogeneous computing with focus on mechanical engineering, PhD dissertation, University of Oslo and Grenoble Institute of Technology, 2009. (Thesis accepted. Defence 2009-05-06.)

[83] R. Holt, LSI technology state of the art in 1968, September 1998.

[84] D. Horn, Stream reduction operations for GPGPU applications, in: *GPU Gems 2*, M. Pharr and R. Fernando, eds, Addison-Wesley, Boston, MA, USA, 2005, pp. 573–589.

[85] L. Howes and D. Thomas, Efficient random number generation and application using CUDA, in: *GPU Gems 3*, H. Nguyen, ed., Addison-Wesley, Boston, MA, USA, 2007, pp. 805–830.

[86] HPCWire, FPGA cluster accelerates bioinformatics application by 5000×, November 2009, available at: http://www.hpcwire.com/offthewire/FPGA-Cluster-Accelerates-Bioinformatics-Application-by-5000X-69612762.html (visited 2009-11-14).

[87] IBM, PowerPC microprocessor family: Vector/SIMD multimedia extension technology programming environments manual, 2005.

[88] IBM, IBM BladeCenter QS22, available at: http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs22/ (visited 2009-04-20).

[89] IBM, Software development kit for multicore acceleration version 3.1: Programmers guide, August 2008.

[90] IBM, Fast Fourier transform library: Programmer's guide and API reference, August 2008.

[91] IBM, 3d fast Fourier transform library: Programmer's guide and API reference, August 2008.

[92] Impulse accelerated technologies, http://impulseaccelerated.com/ (visited 2009-04-28).

[93] H. Inoue, T. Moriyama, H. Komatsu and T. Nakatani, AA-sort: A new parallel sorting algorithm for multi-core SIMD processors, in: *Intl. Conf. on Parallel Architecture and Compilation Techniques*, IEEE Computer Society, Washington, DC, USA, 2007, pp. 189–198.

[94] ISO/IEC, 9899:TC3, International Organization for Standardization, September 2007.

[95] K. Iverson, *A Programming Language*, Wiley, New York, NY, USA, 1962.

[96] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf and K. Yelick, Implicit and explicit optimizations for stencil computations, in: *Workshop on Memory System Performance and Correctness*, ACM, New York, NY, USA, 2006, pp. 51–60.

[97] K. Kennedy, C. Koelbel and H. Zima, The rise and fall of high performance Fortran: an historical object lesson, in: *Conf. on History of Programming Languages*, 2007, San Diego, CA, USA, 7-1–7-22.

[98] Khronos OpenCL Working Group, The OpenCL specification 1.0, 2008, available at: http://www.khronos.org/registry/cl/ (visited 2009-03-20).

[99] M. Kistler, J. Gunnels, D. Brokenshire and B. Benton, Petascale computing with accelerators, in: *Symp. on Principles and Practice of Parallel Programming*, ACM, New York, NY, USA, 2008, pp. 241–250.

[100] T. Knight, J. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. Dally and P. Hanrahan, Compilation for explicitly managed memory hierarchies, in: *Symp. on Principles and Practice of Parallel Programming*, ACM, New York, NY, USA, 2007, pp. 226–236.

[101] C. Koelbel, U. Kremer, C.-W. Tseng, M.-Y. Wu, G. Fox, S. Hiranandani and K. Kennedy, Fortran D language specification, Technical report, 1991.

[102] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas,

M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. Williams and K. Yelick, Exascale computing study: Technology challenges in achieving exascale systems, Technical report, DARPA IPTO, 2008.

[103] M. Krishna, A. Kumar, N. Jayam, G. Senthilkumar, P. Baruah, R. Sharma, S. Kapoor and A. Srinivasan, A synchronous mode MPI implementation on the cell BE architecture, in: *Intl. Symp. on Parallel and Distributed Processing with Applications*, Niagara Falls, ON, Canada, 2007, pp. 982–991.

[104] A. Kumar, G. Senthilkumar, M. Krishna, N. Jayam, P. Baruah, R. Sharma, A. Srinivasan and S. Kapoor, A buffered-mode MPI implementation for the cell BE processor, in: *Intl. Conf. on Computational Science*, Springer-Verlag, Berlin, 2007, pp. 603–610.

[105] E. Larsen and D. McAllister, Fast matrix multiplies using graphics hardware, in: *Supercomputing*, ACM, New York, NY, USA, 2001, p. 55.

[106] P. L'Ecuyer, Maximally equidistributed combined Tausworthe generators, *Math. Comput.* **65**(213) (1996), 203–213.

[107] B. Lloyd, C. Boyd and N. Govindaraju, Fast computation of general Fourier transforms on GPUs, in: *Intl. Conf. on Multimedia & Expo*, Hannover, Germany, 2008, pp. 5–8.

[108] E. Loh and G. Walster, Rump's example revisited, *Reliab. Comput.* **8**(3) (2002), 245–248.

[109] Y. Luo and R. Duraiswami, Canny edge detection on NVIDIA CUDA, in: *Computer Vision and Pattern Recognition Workshops*, June 2008, IEEE Computer Society Press, Washington, DC, USA, 2008, pp. 1–8.

[110] M. Matsumoto and T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, *Model. Comput. Simul.* **8**(1) (1998), 3–30.

[111] M. Matsumoto and T. Nishimura, Dynamic creation of pseudorandom number generators, in: *Monte Carlo and Quasi-Monte Carlo Methods 1998*, Springer-Verlag, Heidelberg, Germany, 2000, pp. 56–69.

[112] T. Mattson, R. Van der Wijngaart and M. Frumkin, Programming the Intel 80-core network-on-a-chip terascale processor, in: *Supercomputing*, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–11.

[113] M. McCool, Data-parallel programming on the cell BE and the GPU using the Rapidmind development platform, in: *gSPx Multicore Applications Conference*, November 2006.

[114] M. McCool, S. Du Toit, T. Popa, B. Chan and K. Moule, Shader algebra, in: *SIGGRAPH*, ACM, New York, NY, USA, 2004, pp. 787–795.

[115] S. McKeown, R. Woods and J. McAllister, Algorithmic factorisation for low power FPGA implementations through increased data locality, in: *Int. Symp. on VLSI Design, Automation and Test*, April 2008, pp. 271–274.

[116] J. McZalpin and D. Wonnacott, Time skewing: A value-based approach to optimizing for memory locality, Technical Report dcs-tr-379, Rutgers School of Arts and Sciences, 1999.

[117] P. Michel, J. Chestnut, S. Kagami, K. Nishiwaki, J. Kuffner and T. Kanade, GPU-accelerated real-time 3D tracking for humanoid locomotion and stair climbing, in: *Intl. Conf. on Intelligent Robots and Systems*, San Diego, CA, USA, 29 October–2 November 2007, pp. 463–469.

[118] P. Micikevicius, 3D finite difference computation on GPUs using CUDA, in: *Workshop on General Purpose Processing on Graphics Processing Units*, ACM, New York, NY, USA, 2009, pp. 79–84.

[119] Microsoft, DirectX: Advanced graphics on windows, available at: http://msdn.microsoft.com/directx (visited 2009-03-31).

[120] Mitronics website, http://www.mitrion.com/ (visited 2009-04-28).

[121] E. Mollick, Establishing Moore's law, *Hist. Comput.* **28**(3) (2006), 62–75.

[122] G. Moore, Cramming more components onto integrated circuits, *Electronics* **38**(8) (1965), 114–117.

[123] H. Neoh and A. Hazanchuk, Adaptive edge detection for real-time video processing using FPGAs, March 2005, available at: http://www.altera.com/literature/cp/gspx/edge-detection.pdf (visited 2009-03-20).

[124] A. Nukada and S. Matsuoka, Auto-tuning 3-D FFT library for CUDA GPUs, in: *Supercomputing*, 2009.

[125] R. Numrich and J. Reid, Co-Array Fortran for parallel programming, Technical report, Fortran Forum, 1998.

[126] NVIDIA, CUDA CUBLAS library version 2.0, March 2008.

[127] NVIDIA, CUDA CUFFT library version 2.1, March 2008.

[128] NVIDIA, CUDA SDK version 2.0, 2008.

[129] NVIDIA, CUDA Zone, available at: http://www.nvidia.com/cuda (visited 2009-03-20).

[130] NVIDIA, Developer Zone, available at: http://developer.nvidia.com/ (visited 2009-09-07).

[131] NVIDIA, NVIDIA's next generation CUDA compute architecture: Fermi, October 2009.

[132] NVIDIA, NVIDIA GeForce GTX 200 GPU architectural overview, May 2008.

[133] NVIDIA, NVIDIA CUDA reference manual 2.0, June 2008.

[134] K. O'Brien, K. O'Brien, Z. Sura, T. Chen and T. Zhang, Supporting OpenMP on cell, *Parallel Prog.* **36**(3) (2008), 289–311.

[135] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu and T. Nakatani, MPI microtask for programming the cell broadband engine processor, *IBM Syst. J.* **45**(1) (2006), 85–102.

[136] OpenFPGA website, http://www.openfpga.org/ (visited 2009-04-28).

[137] OpenGL Architecture Review Board, D. Shreiner, M. Woo, J. Neider and T. Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, 6th edn, Addison-Wesley, Boston, MA, USA, 2007.

[138] Open systemc initiative, http://www.systemc.org/ (visited 2009-04-28).

[139] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone and J. Phillips, GPU computing, *Proc. IEEE* **96**(5) (2008), 879–899.

[140] S. Pakin, Receiver-initiated message passing over RDMA networks, in: *Intl. Parallel and Distributed Processing Symp.*, Miami, FL, USA, April 2008.

[141] Parallel linear algebra for scalable multi-core architectures (PLASMA) project, available at: http://icl.cs.utk.edu/plasma/ (visited 2009-04-20).

[142] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas and K. Yelick, A case for intelligent RAM, *IEEE Micro* **17**(2) (1997), 34–44.

[143] J. Perez, P. Bellens, R. Badia and J. Labarta, CellSs: making it easier to program the cell broadband engine processor, *IBM J. Res. Dev.* **51**(5) (2007), 593–604.

[144] J. Peterson, P. Bohrer, L. Chen, E. Elnozahy, A. Gheith, R. Jewell, M. Kistler, T. Maeurer, S. Malone, D. Murrell, N. Needel, K. Rajamani, M. Rinaldi, R. Simpson, K. Sudeep

and L. Zhang, Application of full-system simulation in exploratory system design and development, *IBM J. Res. Dev.* **50**(2,3) (2006), 321–332.

[145] T. Pock, M. Unger, D. Cremers and H. Bischof, Fast and exact solution of total variation models on the GPU, in: *Computer Vision and Pattern Recognition Workshops*, June 2008, IEEE Computer Society Press, Washington, DC, USA, 2008, pp. 1–8.

[146] Portland Group, PGI accelerator compilers, available at: http://www.pgroup.com/resources/accel.htm (visited 2009-08-05).

[147] T. Purcell, C. Donner, M. Cammarano, H. Jensen and P. Hanrahan, Photon mapping on programmable graphics hardware, in: *EUROGRAPHICS*, Eurographics Association, 2003, pp. 41–50.

[148] RapidMind, Cell BE porting and tuning with RapidMind: A case study, 2006, available at: http://www.rapidmind.net/case-cell.php (visited 2009-03-20).

[149] M. Ren, J. Park, M. Houston, A. Aiken and W. Dally, A tuning framework for software-managed memory hierarchies, in: *Intl. Conf. on Parallel Architectures and Compilation Techniques*, ACM, New York, NY, USA, 2008, pp. 280–291.

[150] Report on the experimental language X10, draft 0.41, Technical report, IBM, 2006.

[151] H. Richardson, High performance Fortran: history, overview and current developments, Technical Report 1.4 TMC-261, Thinking Machines Corporation, 1996.

[152] V. Sachdeva, M. Kistler, E. Speight and T.-H. Tzeng, Exploring the viability of the cell broadband engine for bioinformatics applications, *Parallel Comput.* **34**(11) (2008), 616–626.

[153] M. Saito and M. Matsumoto, SIMD-oriented fast Mersenne Twister: a 128-bit pseudorandom number generator, in: *Monte Carlo and Quasi-Monte Carlo Methods*, Springer-Verlag, Heidelberg, Germany, 2008.

[154] N. Satish, M. Harris and M. Garland, Designing efficient sorting algorithms for manycore GPUs, NVIDIA, NVIDIA Technical Report NVR-2008-001, September 2008.

[155] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan and P. Hanrahan, Larrabee: a many-core $\times$86 architecture for visual computing, *Trans. Graph.* **27**(3) (2008), 1–15.

[156] S. Sengupta, M. Harris, Y. Zhang and J. Owens, Scan primitives for GPU computing, in: *EUROGRAPHICS*, Pergamon Press Inc., Elmsford, NY, USA, 2007, pp. 97–106.

[157] S. Sengupta, A. Lefohn and J. Owens, A work-efficient step-efficient prefix sum algorithm, in: *Workshop on Edge Computing Using New Commodity Architectures*, May 2006, pp. 26–27.

[158] B. Stackhouse, B. Cherkauer, M. Gowan, P. Gronowski and C. Lyles, A 65nm 2-billion-transistor quad-core Itanium processor, in: *Intl. Solid-State Circuits Conf.*, Lille, France, February 2008, pp. 92–598.

[159] O. Storaasli and D. Strenski, Beyond 100$\times$ speedup with FPGAs: Cray XD1 I/O analysis, in: *Cray Users Group*, Cray User Group Inc., Corvallis, OR, USA, 2009.

[160] O. Storaasli, W. Yu, D. Strenski and J. Maltby, Performance evaluation of FPGA-based biological applications, in: *Cray User Group*, Cray User Group Inc., Corvallis, OR, USA, 2007.

[161] D. Strenski, 2009, personal communication.

[162] D. Strenski, J. Simkins, R. Walke and R. Wittig, Evaluating fpgas for floating-point performance, in: *Intl. Workshop on High-Performance Reconfigurable Computing Technology and Applications*, November 2008, IEEE Computer Society Press, Washington, DC, USA, pp. 1–6.

[163] H. Sugano and R. Miyamoto, Parallel implementation of morphological processing on cell/BE with OpenCV interface, in: *Intl. Symp. on Communications, Control and Signal Processing*, St. Julians, Malta, March 2008, pp. 578–583.

[164] J. Sun, G. Peterson and O. Storaasli, High-performance mixed-precision linear solver for FPGAs, *IEEE Trans. Comput.* **57**(12) (2008), 1614–1623.

[165] M. Sussman, W. Crutchfield and M. Papakipos, Pseudorandom number generation on the GPU, in: *Graphics Hardware*, ACM, New York, NY, USA, 2006, pp. 87–94.

[166] Starbridge systems website, http://www.starbridgesystems.com/ (visited 2009-12-04).

[167] S. Swaminarayan, K. Kadau, T. Germann and G. Fossum, 369 tflop/s molecular dynamics simulations on the Roadrunner general-purpose heterogeneous supercomputer, in: *Supercomputing*, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–10.

[168] D. Thomas, L. Howes and W. Luk, A comparison of CPUs, GPUs, FPGAs and masssively parallel processor arrays for random number generation, in: *FPGA*, ACM, New York, NY, USA, 2009.

[169] D. Thomas and W. Luk, High quality uniform random number generation using LUT optimised state-transition matrices, *VLSI Signal Process. Syst.* **47**(1) (2007), 77–92.

[170] Tokyo Tech, November 2008, booth #3208 at *Supercomputing'08*, available at: http://www.voltaire.com/assets/files/Case%20studies/titech_case_study_final_for_SC08.pdf (visited 2009-04-28).

[171] Top 500 supercomputer sites, June 2009, available at: http://www.top500.org/.

[172] Top green500 list, November 2008, available at: http://www.green500.org/.

[173] UPC language specification v1.2, Technical report, UPC Consortium, 2005.

[174] A. van Amesfoort, A. Varbanescu, H. Sips and R. van Nieuwpoort, Evaluating multi-core platforms for HPC data-intensive kernels, in: *Conf. on Computing Frontiers*, ACM, New York, NY, USA, 2009, pp. 207–216.

[175] W. van der Laan, Cubin utilities, 2007, available at: http://www.cs.rug.nl/~wladimir/decuda/ (visited 2009-03-20).

[176] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar and S. Borkar, An 80-tile sub-100-w teraflops processor in 65-nm CMOS, *Solid-State Circuits* **43**(1) (2008), 29–41.

[177] A. Varbanescu, H. Sips, K. Ross, Q. Liu, L.-K. Liu, A. Natsev and J. Smith, An effective strategy for porting C++ applications on cell, in: *Intl. Conf. on Parallel Processing*, IEEE Computer Society, Washington, DC, USA, 2007, p. 59.

[178] Verilog website, http://www.verilog.com/ (visited 2009-04-28).

[179] D. Vianney, G. Haber, A. Heilper and M. Zalmanovici, Performance analysis and visualization tools for cell/B.E. multicore environment, in: *Intl. Forum on Next-Generation Multicore/Manycore Technologies*, ACM, New York, NY, USA, 2008, pp. 1–12.

[180] V. Volkov and J. Demmel, Benchmarking GPUs to tune dense linear algebra, in: *Supercomputing*, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–11.

[181] V. Volkov and B. Kazian, Fitting FFT onto the G80 architecture, available at: http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf (visited 2009-08-10).

[182] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms, in: *Supercomputing*, ACM, New York, NY, USA, 2007, pp. 1–12.

[183] Xilinx website, http://www.xilinx.com/ (visited 2009-04-28).

[184] M. Xu, P. Thulasiraman and R. Thulasiram, Exploiting data locality in FFT using indirect swap network on cell/BE, in: *Intl. Symp. on High Performance Computing Systems and Applications*, IEEE Computer Society, Washington, DC, USA, 2008, pp. 88–94.

[185] L. Zhuo and V.K. Prasanna, High-performance designs for linear algebra operations on reconfigurable hardware, *IEEE Trans. Comput.* **57**(8) (2008), 1057–1071.

[186] G. Ziegler, A. Tevs, C. Theobalt and H.-P. Seidel, GPU point list generation through histogram pyramids, Technical Report MPI-I-2006-4-002, Max-Planck-Institut für Informatik, 2006.

[187] H. Zima, P. Brezany, B. Chapman, P. Mehrotra and A. Schwald, Vienna Fortran – a language specification version 1.1, Technical Report 3, Austrian Center for Parallel Computation, 1992.