High-Performance SDR: GNU Radio and the IBM Cell Broadband Engine

Nick McCarthy[1], Eric Blossom[2], Nate Goergen[1], Tim OShea[1], Charles Clancy[1]

[1]Laboratory for Telecommunications Sciences, University of Maryland
[2]Blossom Research, LLC

Abstract

Software-Defined Radio (SDR) allows the signal processing components of a wireless device to be implemented in a reconfigurable processor. Recent research has focused on using general-purpose processors (GPPs) for performing signal processing in SDRs. While older GPPs lacked the processing power required for implementing many of the modern, commercial, digital waveforms, newer GPPs offer promise.

In this paper, we explore Single Instruction Multiple Data (SIMD) components of the x86, PowerPC, and Cell Broadband Engine (CBE) processors. We also explore the multiprocessing capabilities of the CBE, and its impact on improving the performance of GNU Radio.

1 Introduction

Currently, in order to support high-complexity, high-bandwidth signals, e.g. digital waveforms with signal bandwidths greater than 3 MHz, typically a Field Programmable Gate Array (FPGA) or Digital Signal Processor (DSP) is required [1, 2, 3]. General Purpose Processors (GPPs) simply do not have the needed processing power. Unfortunately, it is generally more difficult to develop signal processing algorithms for FPGAs and DSPs, and even more difficult to perform rapid reconfiguration at run-time. However, by taking advantage of vector processors built into GPPs, we can improve performance. Two key examples of this are the SSE instructions for x86 processors and the Altivec instructions for PowerPC processors.

In this work, we focus on an even higher-performing GPP platform: the IBM Cell Broadband Engine (CBE). The CBE contains a PowerPC processor supporting the Altivec extensions, along with eight Synergistic Processing Elements (SPEs). The SPEs are single-instruction, multiple-data (SIMD) processors capable of performing vector operations in a single clock cycle. The PowerPC and SPEs are all connected to high-speed memory and I/O via the Element Interconnect Bus (EIB).

Recently work has begun to port the GNU Radio SDR to the CBE, and compile critical signal processing components for the SPEs. In this paper we examine various architectures for distributing a single processing flow graph across a multi-processor (MP) environment. We look at tradeoffs between pipelining and random access scheduling approaches to utilizing the SPEs. We outline change to the GNU Radio engine necessary to support MP, and architectures for threading a single-threaded processing engine. We describe the design of a general-purpose process scheduler constructed for the CBE to support tasking jobs to the SPEs. Lastly, we present performance figures for an implementation of the proposed algorithms, and show order-of-magnitude increases in signal processing performance.

The remainder of the paper is organized as follows. Section 2 discusses vector-based processing, in particular the SIMD capabilities of the x86, PowerPC, and CBE. Section 3 describes various multiprocessing approaches, and their relation to the CBE environment. Section 4 discusses GNU Radio's support for these SIMD and MP processors. Section 5 examines possibilities for future work and concludes.

2 Vector-Based Processing

During the late 1990's, an ever increasing demand for numerical processing in rendering graphics, multimedia processing, and physical modeling engines pressured personal computing processors to satisfy the need for repetitive computation on data arrays. Vector operations, specifically those which require a processor to repeat a given basic operation for an entire array of data, are common in signal processing algo-

rithms.

To accomplish vectorized operations on ordinary GPPs, single-instruction processors must resort to software loops in which only single calculations are performed per processor instruction. Since each iteration of such a loop performs the same operation on each element of input data, the operation may be parallelized at the instruction level using SIMD operations. SIMD instructions allow a processor to engage multiple execution units simultaneously and to produce a greater number of output elements per clock cycle. The term *vector instruction* often applies to a SIMD instruction since each SIMD instruction may load, process, and store multiple scalar integer, fixed-point, or floating point items in a single clock cycle.

Historically, SIMD instruction sets existed only for supercomputing platforms, but commercial processor manufactures have taken strides to bring the SIMD model for parallelization to commodity devices through a combination of software and hardware innovations. Instruction sets used for the x86 and PowerPC architectures have benefitted from increased vector processing capabilities over recent years. Starting with MMX, the first SIMD instruction set introduced for the x86 architecture, this platform evolved over time to include the SSE (Streaming SIMD Extensions), SSE2, and SSSE3 instruction sets from Intel, while rival manufacture AMD introduced similar extensions under the 3DNow! trademark.

The IBM PowerPC architecture received a similar SIMD upgrade through the Altivec instruction set. Graphics Processor Units (GPUs) have, for some time, offered a hardware-based engine for SIMD parallelization, and manufacturers have begun to tune these units for use as general-purpose coprocessors, completing the package with documented software interfaces to the devices. The CBE offers an Altivec-enabled primary processor as well as a number of subprocessors able to execute Altivec-like instructions.

With the addition of SIMD capabilities to common personal computing platforms, GPPs have become increasingly able to compete with DSPs, making real-time signal processing on GPP hardware a viable consideration. The ease of programming and reconfiguring programs for GPPs makes advancements in the vector processing capabilities of these platforms notable. Unfortunately, the addition of SIMD instructions can complicate the interface between the programmer and the GPP. While version 3.x of the GNU C Compiler (GCC) included support for SIMD instructions, upgrading compilers and interpreters to make use effectively of SIMD instructions remains a difficult, uncompleted task. Ultimately, we want for compilers automatically to identify vectorizing opportunities and to translate from high-level code to SIMD instruction where appropriate. In order to achieve this task, a language compiler or interpreter must analyze software loop structures and loop unrolling, both complicated processes. GCC did not support auto-vectorization until version 4.x, and the results have proved limited. Currently, programmers seeking to reap the full benefits of SIMD instructions often resort to hand-coded, assembly language programing for select routines to achieve peak performance.

2.1 x86 and PowerPC SIMD Support

Initial SIMD instruction sets for the x86 architecture, namely Intel MMX and AMD 3DNow!, proved to be of limited value in signal processing applications. MMX aliased existing Floating Point Unit (FPU) registers rather than adding new registers and limited vectorization to 64-bit integer operations allowing for configurations of 2 32-bit scalar integers, 4 16-bit scalar integers, or 8 8-bit scalar integer parallelization. Additionally, since the ALU and the FPU shared registers, only one functional unit could operate at a given time, requiring expensive mode switching operations. 3DNow!, similar to MMX in implementation, featured highly valuable floating-point capabilities. These original SIMD extensions featured only vectorized versions of traditional basic arithmetic and logic, a further limitation [4].

SSE, the second generation of SIMD extensions for the x86 architecture, fixed many shortcomings of the first-generation. This implementation introduced 8 new physical vector registers (16 for the Intel64 and AMD64 versions) and extended the size of the registers from 64 to 128-bits. The first SSE implementations could not issue both SSE and FPU instructions in the same clock cycle (known as dual-issue), but they did allow for mixed SSE/FPU instructions without the expensive performance hit of earlier generations. In addition to supporting vectorized versions of basic arithmetic and logic operations, SSE added cache access optimization instructions and advanced floating-point instructions for mathematic reciprocal, square root, reciprocal square root, min, and max, which are of great use in signal processing algorithms [5].

Initially, the SIMD operation sets from Intel and AMD worked exclusively with hardware from the respective manufacturer, but AMD has since adopted the SSE family of instructions for their hardware. Each SIMD extension adds to the capabilities of prior releases, and old instruction sets continue to work.

The second edition of SSE, named SSE2 by Intel, featured improvements to existing instructions and finer cache control. SSE3 and SSSE3 featured instructions to address many byte-alignment difficulties when using SIMD instructions, horizontal arithmetic operations, and primitive assistance for the manipulation of complex numbers. Recently, Intel has released specifications for an SSE4 version. Planned improvements include instructions for performing dot product and absolute difference operations [6].

Concurrent with the rise of the SSE family of SIMD instructions on the x86 platform came the development of Altivec instructions for the PowerPC architecture. In terms of evaluating different platforms for use with SDR projects, Altivec instructions operate in a manner more or less equivalent to SSE instructions, but we consider some of the more notable differences.

Architectures supporting Altivec make available 32 128-bit registers for vector manipulation as opposed to eight such registers. Altivec commands include fairly broad support for horizontal arithmetic, and among the primitives we find a dot product, in addition to the usual reciprocal and reciprocal square root estimations, min, and max. In addition, Altivec supports a byte-shuffling routine with a wide array of uses from managing alignment issues to facilitating complex multiply routines.

Apple's replacement of PowerPC with x86 prompted migration from Altivec to SSE instructions, and this migration in turn prompted comparisons between the platforms. At the time of Apple's conversion in 2005, Altivec running on a G5 processor outperformed SSE3 running on a Xeon 3.4GHz processor [7]. The benchmarks used for this comparison rely heavily on many aspects of hardware, not just those directly related to SIMD instruction sets. Moreover, hardware has evolved since 2005, and we hesitate to take this comparison as evidence for the natural superiority of Altivec-enabled architectures in the present day. The CBE features a PowerPC processor equipped with Altivec instructions [8, 9].

2.2 General Purpose Computation on GPU

The x86 and Altivec families of instruction sets represent productive efforts on the part of industry to squeeze vector-based performance from general-purpose architectures, but the design of these architectures limits their use as a SIMD programming platform. Graphics processing demands, on the other hand, have yielded GPU architectures tuned to achieving high bandwidth SIMD parallelism at the

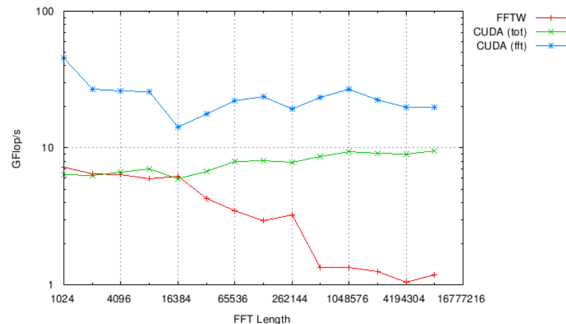

Figure 1: Performance achieved for FFT implementation on a GPU, as a function of the FFT length [10]

expense of the low-latency performance a CPU must offer. Both AMD and NVIDIA have taken strides to unhinge the compute power of their GPU products from the specific tasks of physics engine computations, rasterization, and pixel shading. The AMD Close To Metal (CTM) approach provides an application binary interface and an assembly instruction set to the developer but provides no interface to a high level programming language such as C. The NVIDIA Compute Unified Device Architecture (CUDA) provides a C interface to the hardware, but debilitates performance tuning by keeping the assembly instructions under lock and key. Their results offer flexible DSP horsepower in a commodity computing package, but certain disadvantages apply.

We look closely into the NVIDIA CUDA hardware in order to understand the potential benefits and obstacles to using GPUs as DSPs. NVIDIA organizes as many as 128 processors into groups of eight, each group forming a multiprocessor. Each processor operates at the GPU clock speed, current models featuring clock speeds of no more than 650MHz. An instruction unit feeds each processor within a multiprocessor identical commands for execution. A multiprocessor features 8192 32-bit registers and the processors can perform floating or fixed-point operations on these registers.

Processors draw data from a complicated memory hierarchy. Each multiprocessor hosts 16KB of low-latency, on-chip shared memory. Though limited in size, this memory allows for data sharing among processors of the same multiprocessing unit without cache coherency issues and without performance-throttling load and store times from off-chip memory. In addition, each multiprocessor owns an 8KB set of cached shading memory and another 8KB set of cached constant memory. The remainder of the

device memory comprises uncached global memory and uncached memory local to each multiprocessor. Data passed between the GPU coprocessor and the host processor must traverse a PCI express connection rated at a maximum of 31.25Gb/s [11, 10].

The architecture promises high theoretical performance for many algorithms associated with signal processing. The CUDA approach hard-wired support for highly parallelizable algorithms with little data dependencies between threads. Input data must fit in shared memory in order to achieve full effect, and data-sharing across multiprocessors must occur minimally. Processes bound by I/O stand to take a performance hit, as data flow from the host processor to the GPU coprocessor involves significant overhead.

To examine the potential performance of the CUDA architecture, we consider the discrete Fourier transform (DFT), a workhorse operation common to almost every signal processing project. The Cooley-Turey algorithm for DFT computation in $\mathcal{O}(n \log n)$ time shows a method for parallelizing the operation into as many as $n/2$ threads, where n is the resolution, or number of points, in the DFT output. Sizes of input vectors to the DFT operation commonly fall well below the 16KB threshold when performing real-time signal analysis. Preliminary experimentation from The National Radio Astronomy Observatory suggests NVIDIA's proprietary CUFFT function running on the GeForce 8800 GTX clocks nearly identically to FFTW running on an Intel Core 2 Quad at 2.4GHz with input sizes up to 16384, achieving around 6 Gflop/s for an input size of 2048 [10]. This performance benchmark takes into account data transfer times. These numbers represent an appreciable win for the GPU. Theoretically, the GPU could perform up to 16 DFT computations of size 2048 simultaneously in the same time the host CPU could compute a single pass through the same algorithm, and the host CPU could focus on less parallelizable tasks while the coprocessor executes.

Despite the obvious strengths of the GPU as a coprocessor, not every process in an SDR flow graph would map well to architecture like CUDA. By tying multiple processors to a single instruction unit, the CUDA design greatly complicates branching and branch prediction. Since data differs from processor to processor, code with branches can force different processors down different execution paths. A program executed in parallel on a multiprocessor must run to completion on each thread in order for the program to free the associated processors, so a missed prediction in one thread slows completion for all threads. In an architecture allowing each thread to maintain its own instruction code, branch prediction can save on clock stalls whenever that prediction proves correct more than half the time. If eight processors share the same instruction code, a branch prediction can improve performance only if it proves correct more than 7/8 of the time. Additionally, with the GPU running at a slower clock speed, any code requiring low-latency execution would probably meet with greater success on a host processor than on a GPU coprocessor.

Longer, more complicated SDR code passages might involve more branches, might contain segments requiring low-latency execution, and might involve non-parallelizable subroutines. All these factors suggest the GPU might lend itself better to fine-granularity parallelism restricting the use of the coprocessor to low-level routines more than to coarser models stringing subroutines into larger programs.

2.3 The Synergistic Processing Element

The CBE utilizes SIMD processing in a manner similar to the PowerPC Altivec and x86 SSE models. Each SPE includes 256KB of low-latency local store memory. For each of 128 128-bit registers available to a SPE, load and store operations from local store require 6 cycles at a clock speed of 3.2 GHz. Registers 128 bits in length offer the opportunity to vectorize four 32-bit operations per clock tick, and the assembly instruction set implements this vectorization. Some valuable SIMD instructions available to the SPE include a reciprocal estimator and a reciprocal square root estimator, an element-wise averaging instruction and an absolute difference instruction. The SPE also preserves the Altivec byte-shuffle primitive. Notably missing is support for primitives using horizontal arithmetic such as min, max, and multiply-accumulate.

Compared to the massive parallelization of the GPU, the 128-bit-at-a-time SIMD operations of the SPE appear paltry, but the SPE competes through different means. Meanwhile, the SPE matches the 128-bit vectorized performance of SSE and Altivec instructions but offers some architectural differences, both good and bad. Unlike the x86 and PowerPC platforms, each SPE is a true dual pipeline architecture with load, store, and shuffle instructions in one pipeline, and arithmetic instructions in the other. Well-ordered code can take advantage of this architecture to execute two instructions with every clock tick, effectively doubling the theoretical processing power of the SPE as compared to commodity GPPs. In terms of branch prediction, the SPE offers no hardware support. On the other hand, the SPE does sup-

port limited static branch prediction through software. This branch-prediction utility involves much more overhead for the programmer than the highly-sophisticated, hardware-supported branch prediction in GPP units, and the software prediction model limits the programmer to a single outstanding prediction. As a result, the GPP platforms have an advantage over the SPE in running branch-intensive code. On the other hand, since each SPE operates essentially as its own thread, the existence of branch prediction in software can yield advantages for the SPE versus the GPU coprocessor model. The SPE comes complete with a C/C++ compiler and a comprehensive list of C-language extensions to the assembly instructions. These extensions allow for SIMD vectorization without resorting to assembly, but the compiler does not, in our limited experience, handle dual issues or pipeline stalls with great effect [12].

Firing on all cylinders, the SPE theoretically can post 25.6 Gflop/s. The SDK for the CBE architecture comes complete with a SIMD vectorized, C-language DFT program for radix-2 input sizes. In experiments, we have obtained close to 9 Gflop/s from this program. IBM engineers have reported attaining 98% of the theoretical performance limit using a matrix-multiply program [12].

By hand-coding to keep both pipelines of the SPE architecture at capacity, we might expect to see approximately a threefold increase from the C-language DFT program currently available. At around 20 Gflop/s, our operation would compare roughly to the performance of CUDA's CUFFT measured without considering data movement overhead. Section 3.3 will address the CBE's EIB in greater detail, but we note the EIB's transfer rate indicates we might expect better data-transfer overhead performance from the SPE than from CUDA. Effective use of the EIB will determine, in the end, how much reward we can reap from the impressive raw computing power of the SPE. By minimizing data-transfer overhead to within even a wide approximation of the theoretical limit, we could easily produce a DFT program running at an effective rate of at least twice that of our benchmarks for SSE2-aided GPP processing and CUDA. Our efforts to schedule SPE processes and to use the EIB effectively occupy much of the remaining sections.

We have depicted the SPE as a capable coprocessor limited mostly by the size of its local store memory and by the complexity of performance-tuning its code. Using 128-bits as an atomic unit for register loads and stores imposes alignment considerations for all data manipulation on the SPE. Hand-coding routines in order to achieve dual-issued instructions takes vastly more time and energy than constructing a vectorized C-language program. These considerations leave out entirely the difficulties involved in scheduling data transfers and SPE execution.

The SPE occupies an interesting position in the world of SIMD capable processors and coprocessors. Compared to SSE- or Altivec-enabled GPP processors, the SPE provides roughly equivalent SIMD capabilities with the addition of a dual-issue pipeline and a low-latency local store under the control of the programmer. In return, the programmer must do without effective, automated branch prediction and transparent loads and stores through multiple layers of cache. Compared to the CUDA architecture, the SPE appears overmatched in terms of SIMD power, but high theoretical computing power owing to the dual-pipeline architecture and a high clock rate together with high theoretical data-transfer times make the SPE more than competitive with the GPU on paper.

Of course, reaching for theoretical limits on the SPE requires a great deal of effort compared to simply utilizing the less tunable, more transparent interface CUDA provides to the developer. Moreover the SPE clock rate and branch prediction utilities make it a candidate for coarser parallelization than the GPU coprocessor. It remains important, however, not to overstate the viability of the SPE as a stand-alone processor. Without more automated and effective branch prediction and memory access, the SPE will under perform certain kinds of code execution as compared to SSE or Altivec-equipped GPPs.

3 Multiprocessing

Multiprocessing is rapidly becoming one of the most common and effective strategies for continuing to provide increased usable computational ability from the ever-increasing transistor counts on modern processors, as we continue to follow the Moore's law curve.

While both SIMD and Multiprocessing achieve this goal, they do so in different ways and with different limitations. SIMD introduces new wide-operand instructions into the micro-architecture, in an attempt to reduce required instruction counts. However the vector data operations enabled by SIMD are only effective for some applications, and often require careful, architecture-specific hand coding of assembly instructions or the use of efficient auto-vectorizing compilers to exploit. In contrast, multiprocessing provides an increase in available computational ability by assuming we are able to split our total task into multiple, mostly-independent paths of execution which may be executed simultaneously on multiple

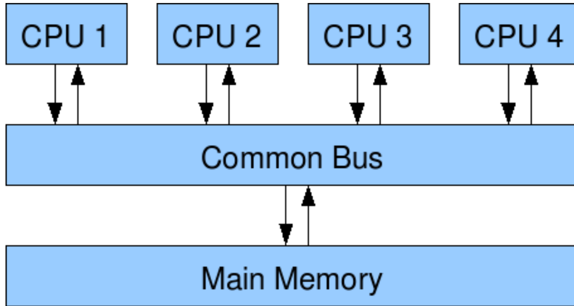

Figure 2: Random Access Processing Model

processing units, instead of time-sharing on a single unit. The primary challenge with this form of multiprocessing has long been determining how to effectively break tasks up over multiple cores, how to ensure their synchronization and cache concurrency, and how to enable all of this without overly encumbering the application developer.

In short, SIMD provides effective small-grained parallelism for a few simple, well-defined, instruction-level tasks which are well suited to accelerating many signal processing primitives. On the other hand, multiprocessing provides effective, large-grained parallelism. This allows separate processes to run simultaneously on distinct resources with carefully planned interactions. This is well suited to hosting separate, independent, signal-processing tasks. Both of these strategies provide effective speedup, and both must be utilized to maximize the potential of current-day architectures.

There has been much research exploring how to effectively organize multi-core and many-core architectures efficiently for various tasks, and they can roughly be broken down into random-access and pipelined processing.

3.1 Random Access Processing

In a random-access multi-processor architecture, tasks running on each processor all access a common main memory controller to retrieve their inputs and write back their outputs. There is no notion of data flow between processors; all access occurs from memory back to memory. This architecture provides the feature that each processor is essentially the same from the task's standpoint. Any task may be scheduled on any of processor, and its job of reading and writing data to main memory does not change. This makes it an ideal architecture if we are interested in providing a generic platform onto which we schedule a random selection of independent tasks.

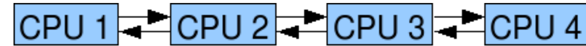

Figure 3: Pipelined-Processing Model

It is ideal then that commercial desktop processors have followed this model, allowing generic applications to continue operating unmodified on concurrent processors, with little complication. There are obvious benefits to this highly-flexible architecture in the case of SDR where we are interested in dynamically building waveforms out of a number of component tasks which will only be determined at runtime.

However, this blind random access model provides some challenges in scaling. Since all processors must access main memory over some common bus, we are limited by both memory access speeds and common bus throughput. This issue is compounded if processors have traditional caches governed by automatic caching algorithms which select blocks to store locally without the knowledge of the running task. In this case, each processor must also broadcast cache invalidations over the common bus to other processors, ensuring that stale copies of the same memory block are not used.

3.2 Pipelined Processing

In the pipelined approach, processors are generally laid out in a way that one processor may pass information to another over a direct or switched interconnect. By connecting a chain of processors in this fashion, data rates achieved between processors may be extremely high since there is no contention for a common bus.

There have been numerous architectures proposing different static or switch layouts of processors, I/O devices, and memories. Many of the approaches have shown promising performance results, but in virtually all of these cases, the problem of mapping the desired well-defined application onto this architecture was done manually. In the case of a highly-flexible SDR in which the appropriate waveform and all of its components may not be known until run-time, and may need to be updated repeatedly, the constant optimal re-mapping of tasks onto a purely pipelined architecture is challenging problem.

3.3 Multiprocessing on GPPs

Since the 1960s, mainframe computers have supported multiple GPPs. Throughout the 1980s and 1990s, high-end servers utilized multiple processors.

For example, Sun Microsystems's top-end server in the late 1990s utilized 64 Ultrasparc processors. However it wasn't until Pentium II and the Athlon MP systems that multi-processor systems began showing up on desk tops in significant quantities. With the Intel Core 2 Duo and AMD Athlon64 X2 releases in 2006 and 2007, respectively, the majority of new desktop and laptop PCs have multiprocessing built-in. Similarly, PowerPC 970MP was released in 2005, and is used in Apple G5 processors and in the IBM JS21 blade server.

With multiple processors, users can run multiple applications simultaneously, and see immediate performance gains. Modern operating systems involve running 100s of independent applications simultaneously, which can each be assigned to one of the multiple processors. However, if a single application wants to take full advantage of a multi-processor system, it must be multi-threaded. Modern operating systems allow each thread of a threaded application to execute on an independent processor.

In SDR applications, this can manifest itself in different ways. An approach taken by the Software Communications Architecture (SCA) is to run each signal processing block within a separate application, and use CORBA to perform communication between them. GNU Radio, as discussed in the next section, will use a thread pool and assign processing blocks to specific threads.

Running each signal processing block in its own thread is the easiest approach to threading a system, but utilization of a thread pool gives the engine more fine-grained control over block execution, allowing for better overall performance tuning for a specific architecture.

3.4 CBE Design Considerations

The IBM CBE provides 1 PowerPC core (the PPE) and 8 SPEs (Figure 4) interconnected with each other and the main memory via the EIB. Each SPE supports SIMD execution, as described in Section 2. Generally, applications can either be written to primarily run on the PPE and offload compute-intensive sections to the SPEs (i.e. the random access model), or be designed to pipeline execution through the SPEs. There are two basic approaches for pipelined execution: one where code is loaded on to the SPEs, and data is then DMA'd from one to the next, being processed as it goes; or alternatively where chunks data are DMA'd to SPEs, and then code is moved on and off the SPEs.

While the EIB, does provide a common bus from which all devices may access main memory, it is not a traditional random access model. The local cache blocks on each processor, referred to as local store, are managed manually by moving blocks in and out only as instructed by the application, and there is no need for any kind of cache concurrency traffic on the EIB, allowing the bus to scale effectively to a large number of cores.

However, the EIB is not a broadcast bus at all, it is a double-wide, bi-directional, pipelined bus in which data is transferred one hop per two processor cycles in the shortest path from source to destination around a ring. Transfers may be pipelined, so we can achieve the full transfer rate of 25.6 Gbytes/s along any non-overlapped path on the ring. This design enables it to function as a high-speed ring of pipelined processors if desired, with the appropriate software mappings. This flexibility is ideal for a highly-configurable SDR platform, providing the potential for the scheduler to simply lay out independent tasks on arbitrary processor cores working from main memory, or to intentionally take existing logical flow graph chains and map them to physical processors located next to each other along the ring to ensure that their communications have low contention from each other, and provide both high-throughput and low latency.

This hybrid architecture is able to achieve many of the benefits from both the random access and pipelined multiprocessing models and provides an ideal platform for building highly flexible signal processing chains.

4 GNU Radio Implementation

In this section we describe current GNU Radio support for both SIMD and multiprocessing, and provide results from specific benchmarks on the IBM CBE.

4.1 Support for SIMD

GNU Radio currently takes advantage of processor-specific SIMD instructions in the lowest level filter kernels. These were selected for optimization based on profiling common workloads. Example kernels include dot products, with all mixes of input types and output types including float, complex¡float¿, and short.

Each SIMD interface is defined by an abstract C++ class. For each supported architecture and SIMD instruction set there exists a concrete class that calls out to hand-coded assembly. There is also a generic C implementation that serves as a fallback for architectures that aren't currently supported, and as a reference implementation for the SIMD versions. The selection of which concrete classes to use is

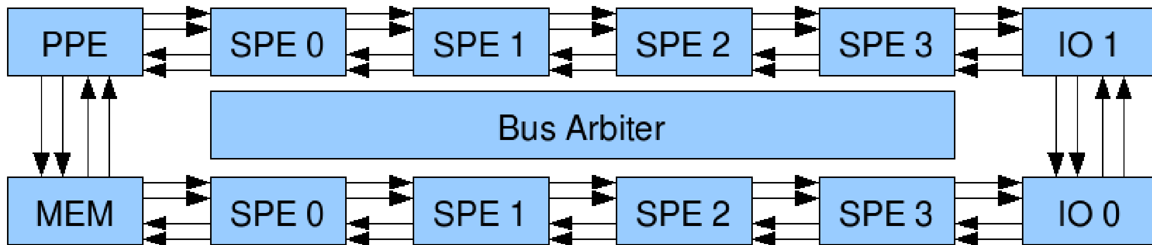

Figure 4: Logical Diagram of Cell Broadband Engine

partitioned between compile-time and run-time. At compile-time all concrete classes that could run on the given architecture are compiled and linked into a table. At run-time, based on the hardware detected, the fastest implementation is selected and returned by a factory method.

GNU Radio has SIMD code for the 3DNow!, SSE, and SSE2 instruction sets on x86 and x86_64 platforms, and limited support for Altivec on the PowerPC.

We are currently investigating additional SIMD routines for frequent operations, such as point-wise multiply of arrays, fast single-precision transcendentals, and numerically controlled oscillators.

4.2 Support for MP

At this time, GNU Radio does a poor job of exploiting the capabilities of commodity multi-processor machines. GNU Radio currently only uses multiple threads if it can localize independent subgraphs in the processing graph. Each subgraph, for example implementing the transmitter and receiver chain of a radio transceiver system, is currently run in an independent thread.

Work is underway to extend the GNU Radio scheduler so that it is cognizant of the number of cores available and to have it exploit them effectively. Because of the data flow model used in GNU Radio, each signal processing block becomes an opportunity for parallel execution. No changes to the user visible API will be required. Existing applications will just run faster when additional cores are available. The approach being pursued is implementation of a thread pool that GNU Radio's scheduler can use to implement the `work()` function of various processing blocks simultaneously.

4.3 Support for CBE SIMD and MP

Although it is possible to build a software radio that has fixed computational requirements (e.g., an embedded HDTV receiver), the SDRs and cognitive radios that we're interested in have time-varying workloads.

Some problems are trivially parallelizable and can be statically mapped across computational elements. Many image processing applications fall into this category.

The time-varying nature of the SDR problem argues against a static partitioning of the computation across the available processing elements. In the case of the CBE the vast majority of the computational resources are contained in the SPEs. As pointed out in Section 2.3, SPEs have a relatively small amount of local store which must hold the instructions and data that are to be operated upon.

The PowerPC element in the CBE, though clocked at 3.2 GHz, is very stripped down and has moderate performance. In order to make best use of the CBE, as much work as possible needs to offloaded onto the SPEs. To effectively utilize the CBE, parallelism must be extracted at two levels: pieces of work that can be handed off to the SPEs for execution in parallel, and SIMD parallelism on the SPEs realized through adroit use of the SPE instruction set.

4.3.1 gcell

gcell is a generic offload mechanism and asynchronous RPC mechanism for handing off potentially small jobs for execution on SPEs. It consists of a small kernel (∼10 KB) that runs on the SPE, and a PPE-based library that allows jobs to be asynchronously submitted for execution on any of the available SPEs.

At initialization time, PPE code creates the manager, telling it the number of SPEs that it should allocate and manage (1 to 16 on current hardware) and the code that should run on the SPEs. At this time, all SPEs run the same code, though in the future we expect to dynamically load code into the SPEs, as required, using some variation on position-independent code and a least-recently-used (LRU) plug-in manager. Overlays would work too if you can live with

their constraints.

After the one-time initialization is performed, PPE user code allocates and fills in job descriptors and submits them to the job manager. Fundamentally the job descriptor contains a `proc_id` that identifies the code that is to be executed on the SPE along with descriptions of the input and output arguments that are to be passed. *gcell* handles the details required to achieve high-throughput DMA on the CBE. The user need only provide the EA address and length of each argument. *gcell* arranges for all transfers to be cache aligned for maximum throughput, and handles all corner cases for arbitrary length transfers.

Jobs are submitted into a global work queue. The kernel running on the SPEs extracts a job from the global work queue when it is ready for new work, DMAs any input arguments into the SPE local store, and then calls the designated procedure. Once the procedure completes, the SPE kernel double buffers any arguments that need to be written back to EA memory and asynchronously notifies the PPE job manager using a combination of DMA to EA memory and non-blocking writes to an interrupt mailbox.

User code can asynchronously submit as many jobs as it likes and then wait for all or any of them to complete. With little effort, this allows parallelism to be extracted and realized. Using GNU Radio as an example, the FFT signal processing block is often called with sufficient input data and output buffers to allow multiple FFTs to be evaluated. Instead of iterating over the vectors of data serially, calling the underlying FFT primitive on each one, the *gcell* version of the FFT signal processing block iterates over the vectors and submits an asynchronous job for each vector, then waits for them all to complete.

4.3.2 SPE SIMD Support

SIMD support on SPEs is quite similar to that seen on GPPs: vectorizing compilers are not yet generally effective, coding in C or C++ using intrinsics that map more or less 1-to-1 to instructions is a viable option, but for the ultimate in performance hand-coded assembly language is required.

Unlike contemporary superscaler GPPs with complex cache hierarchies, getting near-theoretical performance out of the SPEs at the assembly level is relatively straight forward. All directly addressable memory is effectively L1 cache and has a fixed and known latency (6 clock cycles). The SPE has a dual-issue pipeline with simple rules that assign instruction classes to one pipe or the other. Instruction execution time is completely deterministic and easy to predict.

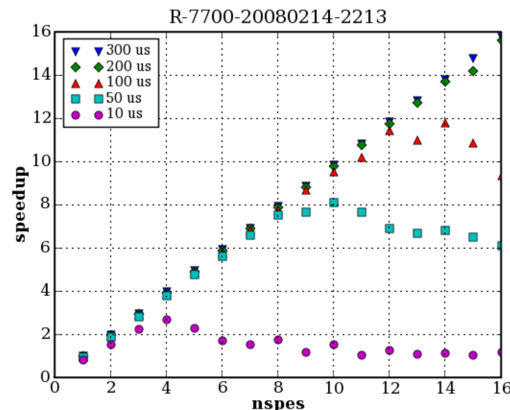

Figure 5: Speedup as $f(nspes, \texttt{t_delay})$

4.4 Performance Analysis

gcell has been benchmarked on Sony Playstation 3s (PS3s) and IBM QS21 blade servers running GNU/Linux. When running on top of the Sony hypervisor, PS3s have 6 SPEs available to use. The IBM QS21 2-way blade server has 16 SPEs.

Our benchmark consists of an application that submits and waits for a total of $njobs$, typically 500,000, where each job busy waits for a specified period of time (`t_delay`) on the SPE before returning. There are a maximum of 64 jobs in flight at any given time. `total_elapsed_time` is the wall clock time between the time the first job is submitted and the final job completes. For each combination of number of SPEs used and `t_delay` we compute a speedup factor (total useful work divided by the total elapsed time):

$$speedup = \frac{njobs * \texttt{t_delay}}{\texttt{total_elapsed_time}} \quad (1)$$

Results are plotted in Figure 5. When using between 1 and 8 SPEs we see near linear speedup for jobs with `t_delay` ≥ 50 μs. Jobs with `t_delay` ≥ 100 μs are near linear out to 12 SPEs, while jobs with `t_delay` ≥ 200 μs are near linear all the way to 16 SPEs.

The current bottleneck is believed to be contention on the lock protecting the shared work queue. We have several ideas on how to mitigate this. We believe that near-linear speedup out to 16 SPEs for jobs with `t_delay` ≥ 25 μs should be possible with a bit of additional effort.

5 Conclusion

In this paper we've discussed both SIMD and multiprocessing aspects of high-performance SDR implementations on GPPs. Through the various approaches described, we show how we can harness components in GPPs to give SDR performance previously only thought to be possible with FPGAs.

We believe that using *gcell* we can implement a real-time transceiver system that supports bandwidths on the order of 20 MHz. This would allow SDR-based implementations of wideband, high-rate, commercial waveforms on commodity GPP hardware.

Current work is also underway with GNU Radio to implement a Virtual Radio Kernel (VRK) within the GNU Radio scheduler. This would allow SDR processing blocks to exist across multiple, independent machines, allowing for GPP, cluster-based processing for GNU Radio-based SDR systems.

References

[1] M. Cummings and S. Haruyama, "FPGA in the software radio," *IEEE Communications Magazine*, 1999.

[2] C. Dick and H. Pederson, "Design and implementation of high-performance FPGA signal processing datapaths for software defined radios," Embedded Systems Conference, 2001.

[3] J. H. Reed, *Software Radio: A Modern Approach to Radio Engineering*. Prentice Hall PTR, 2002.

[4] I. Berkeley Design Technology, *DSP on General-Purpose Processors*. Technical Report, 1997.

[5] B. Patwardhan, "Introduction to the streaming SIMD extensions in the Pentium III: Part i," *Dr. Dobb's Journal*, 2001.

[6] R. M. Ramanathan, "Extending the world's most popular processor architecture: New innovations that improve the performance and energy efficiency of intel architecture." Intel White Paper, 315383-001US, 2006.

[7] P. Software, "macintel faster than Altivec and the Codewarrior?." http://www.pixelglow.com/stories/macintel-faster-than-altivec/, June 2005.

[8] P. Seebach, "Unrolling Altivec, part 1: Introducing the PowerPC SIMD unit." http://www-128.ibm.com/developerworks/library/pa-unrollav1/, March 2005.

[9] E. Rutlege, "Altivec extensions to the portable expression template engine," High Performance Embedded Computing, 2002.

[10] P. Demorest, "GPU benchmarking." http://www.cv.nrao.edu/∼pdemores/gpu/.

[11] NVIDIA, "CUDA compute unified device architecture programming guide version." Version 1.1, November 2007.

[12] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation." http://www.ibm.com/developerworks/power/library/pa-cellperf/.