GCELL – AN SPE SCHEDULER AND ASYNCHRONOUS RPC MECHANISM FOR THE CELL BROADBAND ENGINE

Eric A. Blossom (Blossom Research, LLC, Reno, NV, eb@comsec.com)

ABSTRACT

The Cell Broadband Engine is one of today's most potent floating point signal processing platforms. Effectively a 9-way heterogeneous SIMD system on a single die, the Cell is capable of 200 GFLOPS peak.

In some domains, such as image processing, the partitioning of work across the SPEs may be trivial. However, the SDR and cognitive radio realms often see time-varying work loads, with varying degrees of available parallelism. *gcell* solves the problem of efficiently distributing potentially small tasks across the SPEs by using a distributed SPE-centric scheduler that pulls work to SPEs as they become available. In addition *gcell* provides high-performance DMA of arguments to and from the SPEs, task completion notification to client processes, and binding and rendezvous between PPE and SPE code. Benchmarks show near linear speed-up from 1 to 16 SPEs on 2-way Cell blades.

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) is required [1, 2, 3]. Digital Signal Processors (DSPs) and 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 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 examples of this are the SSE family of instructions for the x86 processors and the VMX (aka AltiVec) instructions for the PowerPC processors.

In this work, we focus on an even higher-performing GPP platform: the IBM Cell Broadband Engine (CBE) [4]. The CBE contains a PowerPC processor (PPE) supporting VMX 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 PPE and SPEs are connected to each other, 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 to compile critical signal processing components for the SPEs. As part of that effort, we have created the *gcell* library[5, 6]; *gcell* efficiently maps the time-varying workloads typical of SDR and cognitive radio across the Cell SPEs. As described in sections 2 and 3, the Cell is capable of outstanding integer and single-precision floating point performance, but its architecture provides a number of challenges and opportunities. *gcell* provides an easy to use interface that handles the dynamic assignment of jobs to SPEs, decouples the expression of parallelism in the user application from the number of SPEs that are being managed, achieves near-linear speedup from 1 to 12 SPEs for jobs that require ≥ 100 μs to execute, and handles all the details required to ensure optimal DMA transfers between the PPE and SPEs regardless of alignment and size.

Section 2 provides an overview of the Cell Broadband Engine. Section 3 highlights the key issues in getting good performance out of the Cell. The *gcell* API is covered in Section 4. Section 5 explains how *gcell* is implemented on the PPE and SPEs, and outlines the key data structures and techniques that allow us to achieve good performance. Section 6 discusses performance. Section 7 looks at related work, while 8 discusses future directions.

2. CELL BROADBAND ENGINE

The CBE processor is a single-chip multiprocessor with nine processor elements operating on a shared, coherent memory, as shown in Figure 1. The first processor element, the PPE, is a traditional 64-bit PowerPC processor core supporting the VMX multimedia extensions. The PPE is a dual-issue, in-order-execution design, with two-way simultaneous multithreading. Many instructions are microcoded or not fully pipelined [7]. Compared to an equivalently clocked PowerPC 970MP, the PPE is quite slow.

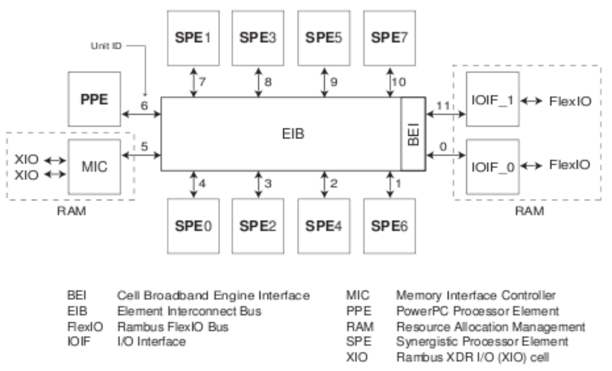
SPE1
SPE3
SPE5
SPE7
Unit ID
PPE
EIB
BEI
MIC
XIO
XIO
RAM
IOIF_1
FlexIO
IOIF_0
FlexIO
RAM
SPE0
SPE2
SPE4
SPE6
BEI Cell Broadband Engine Interface
EIB Element Interconnect Bus
FlexIO Rambus FlexIO Bus
IOIF I/O Interface
MIC Memory Interface Controller
PPE PowerPC Processor Element
RAM Resource Allocation Management
SPE Synergistic Processor Element
XIO Rambus XDR I/O (XIO) cell

Figure 1: Cell Broadband Engine Overview [7]

The second type of processor element, the SPE, is optimized for high-performance SIMD processing. Each SPE includes 256 KB of low-latency local store (LS) memory. For each of the 128 128-bit registers available to an SPE, load and store operations from local store require 6 cycles to complete at a clock speed of 3.2 GHz. 128-bit registers allow four 32-bit operations per clock tick. With the exception of double-precision floating point, all instructions are fully pipelined and can issue on each cycle. (The IBM QS22 fully pipelines double-precision.) The SPE contains two pipelines, the “even” and “odd” pipes. Instructions are statically assigned to one of the two pipes: integer, floating point, and logical instructions to the even pipe; loads, stores and shuffles to the odd pipe. If instructions are appropriately aligned in memory, all data dependencies are fulfilled, and no structural hazards exist, the SPE can issue two instructions per clock, one to the even pipe and one to the odd pipe. This gives a single-precision operation rate of

3.2e9 cycles/s * 4 FLOPS/cycle = 12.8 GFLOPS per SPE

or 102.4 GFLOPS per CBE. When executing the single-precision fused multiply-accumulate instruction, an SPE executes 8 FLOPS per cycle, giving a peak rate of 25.6 GFLOPS per SPE or 204.8 GFLOPS per CBE.

2.1 DMA

Although loads and stores on the SPE can only access the local store, an SPE can transfer data between its LS and the main memory of the system (or the LS of another SPE) using its associated DMA controller, the Memory Flow Controller (MFC). Unlike most DMA controllers, the MFC deals in “Effective Addresses” (EA), IBM terminology for something very much like a virtual address. Code running on the SPE executes in user mode and may directly program the MFC for “puts” or “gets” from LS to EA space. Each MFC supports up to 16 simultaneous DMA operations, and each operation is associated with a user-specified 5-bit tag. By default the hardware is free to reorder transfers to maximize throughput. When necessary, you can order the transfers that share a given tag by using the *fence* or *barrier* DMA modifiers.

Besides DMA, the CBE provides two additional facilities for interprocessor communication: mailbox channels and signal notification channels. Each SPE has 2 outbound mailboxes and 1 inbound mailbox. Each mailbox implements a short queue of 32-bit messages. Signaling channels are similar, though they have a maximum depth of one and writes to them may be configured to overwrite or bitwise-or the current contents.

2.2 EIB

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 store on each SPE is managed manually by moving blocks in and out only as instructed by the application; there is no need for any kind of cache concurrency traffic on the EIB. This allows the bus to scale effectively to a large number of cores.

Moreover, the EIB is not a broadcast bus at all; it's 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 GB/s along any non-overlapped path on the ring [8]. With the appropriate software mappings, this design enables the CBE to function as a high-speed ring of pipelined processors. This flexibility is ideal for a highly-configurable SDR platform. The scheduler can simply lay out independent tasks on arbitrary processor cores working from main memory, or — for increased throughput — map the computation and data flows onto adjacent physical processors on the ring, bypassing the PPE and main memory altogether.

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

3. CBE CHALLENGES

Existing POSIX code can be trivially ported to the CBE by recompiling for the PowerPC. However, this leaves you with an implementation that runs, but has mediocre performance. To extract any kind of reasonable throughput from the CBE, performance critical code must be moved to the SPEs. There are several ways hybrid PPE-SPE programs can be structured. One is to write the bulk of the application to run on the SPEs and then do only a trivial amount on the PPE. Variations on pipelining are possible, where data is moved directly from SPE to SPE without going through main memory or involving the PPE. This has high throughput and low latency, but load balancing is critical and SPE to SPE transfers are hard to orchestrate with dynamically varying workloads. Another variation is to keep the data on the SPEs, then DMA the code to the them as needed. Probably the most common arrangement for existing programs is the "off-load" model, where computationally intensive operations are handed off to SPEs for execution, while the bulk of the program runs unmodified on the PPE.

In our experience, obtaining anything near theoretical performance on an SPE requires coding in SIMD intrinsics (C or C++ built-in functions that map 1-to-1 to SPE SIMD instructions) or for ultimate performance, coding directly in SPE assembly. (Coding in assembly gives you explicit control over instruction alignment and pairing which are essential to achieve dual-dispatch in inner loops.)

The SPE also requires explicit branch prediction using "branch hint" instructions. A non-hinted taken branch, or an incorrectly hinted branch results in a 19 cycle stall. The other bind is that to be effective, a hint must be issued about 15 cycles before the branch instruction is executed. Finally, there can be at most a single branch hint active at any time. The end result of this is that your inner loop — if correctly hinted — flies, while outer loops pay the not-hinted taken-branch penalty.

The good news about coding for the SPE is that it has a straight-forward execution model with a few easily understood rules, deterministic timing of instructions, and 128 registers that make it easy to hide load and execution latencies. Coding for the SPE feels very much like running on "bare metal."

4. API

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

There are two primary abstractions: the *job manager* and the *job*. At initialization time, user code creates the manager, specifying the number of SPEs that it should allocate and manage (1 to 16 on current hardware) and the code that should run on them.

Jobs are described by a small structure that tells the *gcell* runtime the procedure that is to be invoked on the SPE and the arguments that are to be passed into and out out of it. The procedure is represented in the job descriptor by an opaque handle. The job manager **`lookup_proc`** method returns a handle given a string that names the procedure. Depending on their size, input and output arguments are handled either as direct arguments or EA arguments. Direct arguments are scalars such as integers, floats, or complex numbers, and are contained in the job descriptor itself. EA arguments are used to pass vectors or other large data structures between the PPE and SPE. For each EA argument, you specify its address, length and whether it is to be copied to or from the SPE.

Once a job descriptor is filled in, you simply call the manager's **`submit_job`** method to queue the job for execution on one of the SPEs. **`submit_job`** returns immediately after queuing the job. When you're ready to wait for a job or set of jobs to complete, call the **`wait_job`** or **`wait_jobs`** method, indicating whether you want to wait for *any* or *all* of the specified jobs to complete before returning.

As a concrete example of usage, GNU Radio's multiprocessor-aware scheduler allows each signal processing block to freely submit and wait for gcell jobs. Beyond the per-block parallelism, it is often possible to expose additional parallelism with minimal effort. For example, the FFT signal processing block's **`work`** method is often called with sufficient data items 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 block iterates over the vectors, submits a job for each one, then waits for them all to complete.

5. INTERNALS

To implement the API, *gcell* must handle 7 primary tasks: submitting jobs for execution, assigning jobs to SPEs, DMA'ing input arguments to the SPE, running the SPE procedure, DMA'ing output arguments back to the PPE, notifying the PPE that the job is done, and notifying the user code that the job is done.

5.1 Job Submission

Job submission is low-overhead. A brief sanity check is performed on the job descriptor, then it's added to the end of a global queue that's shared between the PPEs and the SPEs. The queue and the job descriptors live in EA memory; the queue is protected by a spin lock that's implemented using the load-with-reservation and store-conditional atomic primitives available on the PPE and SPE.

5.2 Assignment of Jobs to SPEs

Idle SPEs pull jobs to themselves by polling the global queue. As part of the dequeuing operation, the SPE ends up with a local store copy of the job descriptor. Contention is minimized by using randomized exponential back-off, where the back-off parameters are a function of the number of SPEs being managed. (We've also tried using the *Lock-Line Reservation Lost* event on the SPE, but contrary to our expectations, found that it didn't perform as well as exponential back-off.)

5.3 Input Arguments

The direct input arguments are contained in the job descriptor, thus we already have them. The input EA arguments, however, must still be DMA'ed into LS from EA memory. The MFC is capable of very slow 1, 2, 4 or 8 byte DMA transfers, or relatively fast transfers that are 16-byte aligned and have lengths that are a multiple of 16-bytes. For the highest throughput, however, DMA transfers must be 128-byte aligned, with lengths a multiple of 128-bytes. 128-byte aligned transfers (the size of a cache line on the CBE, and the transfer unit on the EIB) achieve twice the throughput of 16-byte aligned transfers that are not also 128-byte aligned. Since the *gcell* kernel on the SPE is responsible for LS buffer allocation for the incoming arguments, it can always satisfy the 128-byte alignment and length constraints by allocating a buffer no greater than 127 bytes larger than the user specified length, starting the DMA at the user provided EA address rounded down to a multiple of 128, and rounding up the net transfer length to a multiple of 128. (Although this may copy additional bytes on either side of the user specified EA argument, the extra bytes go into the padding allocated as part of the alignment procedure, and do not impact the user SPE code that is about to be called.) In a similar manner, LS buffers are allocated for output EA arguments.

5.4 Invoking the SPE Procedure

To understand how we locate and call an SPE procedure, it's useful to understand the mechanism used to register SPE procedures with *gcell*. All SPE routines callable by *gcell* have the same signature and are registered using the **`GC_DECLARE_PROC`** macro:

```
#include <gc_declare_proc.h>

static void
my_spe_proc(const gc_job_direct_args_t *input,
            gc_job_direct_args_t *output,
            const gc_job_ea_args_t *eaa)
{
  // compute something wonderful...
}

GC_DECLARE_PROC(my_spe_proc, "my_spe_proc");
```

Behind the scenes, **`GC_DECLARE_PROC`** initializes a data structure that contains the name of your procedure and a pointer to it. The trick is that the structure is contained in a special section of the SPE ELF object file called **`.gcell.proc_def`**. When you link your SPE object files together to form an SPE executable, all the structures initialized by **`GC_DECLARE_PROC`** end up in a contiguous array in the initialized data of the executable. This array of structures is loaded into the SPE along with the rest of the executable. The PPE has access to the table as well, since it has a handle to the SPE program. The job manager **`lookup_proc`** method searches the table for the name you specified, returning the index if found. On the SPE side, the *gcell* kernel knows the address of the table in local store. Thus, the SPE procedure is called by indexing into a table of function pointers.

5.5 Output Arguments

Once the SPE procedure returns, any output EA arguments must be copied from LS back to EA memory. These transfers are double-buffered and require a bit more finesse than the input case, since we can't write outside of the bounds of the EA argument buffer. We correctly handle EA arguments of any alignment and size, but performance is better if your arguments are 16-byte aligned and a multiple of 16-bytes long.

5.6 Notifying PPE of Job Completion

The PPE needs to be told when an SPE completes a job. Our first implementation simply wrote the completed *job id* into an interrupt mailbox connected to the PPE. We noticed however, that the SPEs spent a lot of time blocked in the mailbox write waiting for the PPE to empty its side of the mailbox. The current implementation uses two double buffered arrays of *job ids* in EA space per SPE and an LS working copy. When a job completes, the *job id* is written into the next free slot in the working copy. When we can write a message without blocking, or the working copy is full, we DMA the working copy to EA memory and send a message to the PPE telling it which array to check. This decouples the SPE and PPE, allowing the SPE to dequeue and process additional jobs independent of the load on the PPE.

5.7 Notifying User of Job Completion

The final task on the PPE side is to keep track of the completed jobs and to wake clients blocked in the **`wait_job`** or **`wait_jobs`** methods. The job manager uses an event handling thread to process job completion messages from the SPEs and to notify waiting clients. The coordination between the client threads and the event handler takes place through per-client data structures implemented in thread local storage.

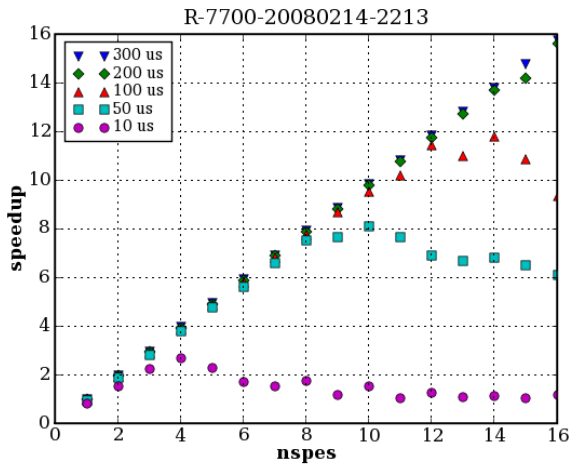

Figure 2: Speedup as *f(nspes, t_delay)*

6. PERFORMANCE

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

Our benchmark consists of an application that submits and waits for a total of *njobs*, typically 500,000, where each job 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):

$$speedup = \frac{njobs \times t_{delay}}{total\, elapsed\, time}$$

Results are plotted in Figure 2. 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.

7. RELATED WORK

The Sequoia project at Stanford is working on a general solution to creating portable programs that are explicitly aware of the memory hierarchy[9, 10, 11]. Starting with an abstract machine model that assumes that there are multiple independent memory spaces exposed to the program, they are working on a C-like program language, compiler and runtime system. They have implemented versions of the system for Cell processors and distributed memory clusters.

Mercury Computer Systems offers “MultiCore Framework” (MCF), basically an API and set of tools for the Cell processor that is primarily oriented toward workloads that can be partitioned by distributing “tiles” of the full data set to be processed across the available SPEs [12]. This works well for large, highly regular problems such as those frequently found in imaging, seismology, or radar. Outside of the ability to load an entire new application into one or more SPEs, there isn't much support for handling dynamically varying work loads.

8. FUTURE DIRECTIONS

We plan to continue to investigate the causes of the scaling limits we're seeing, and if possible, work around them.

At this time all SPEs run the same code. As we continue to move additional performance critical functions into the SPEs, 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 are an option too.

If required by our users, we will add a facility to partition and handle larger data sets than will fit in the local store. This would be built on top of the existing interface, and provide a way to handle problems amenable to the “tiling” or “strip mining” approach.

We also plan to investigate strategies for scheduling jobs using direct SPE to SPE communications, thereby bypassing the PPE and taking better advantage of the peak capacity of the EIB.

9. CONCLUSION

In this paper we've taken a look at the Cell Broadband Engine, the characteristics that make it attractive for SDR, the challenges in using it, and the *gcell* library as a way to exploit it. 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 hardware.

10. 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] International Business Machines Corporation, Sony Computer Entertainment Inc., Toshiba Corporation, *Cell Broadband Engine Architecture, Version 1.01*, 2005, 2006.

[5] http://gnuradio.org/trac/browser/gnuradio/trunk/gcell
[6] http://gnuradio.org/trac/wiki
[7] International Business Machines Corporation, Sony Computer Entertainment Inc., Toshiba Corporation, *Cell Broadband Engine Programming Handbook, Version 1.1*, 2006, 2007.
[8] M. Kistler, M. Perrone, and F. Petrini, “Cell Multiprocessor Communication Network: Built for Speed,” Micro, IEEE, Vol. 26, No. 3, pp. 10-23.
[9] Fatahalian, K., Knight, T. J., Houston, M., Erez, M., Horn, D. R., Leem, L., Park, J. Y., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing.*
[10] Knight, T. J., Park, J. Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W. J., and Hanrahan, P. Compilation for Explicitly Managed Memory Hierarchies *PPoPP 2007*
[11] http://sequoia.stanford.edu
[12] B. Bouzas, R. Cooper, J. Greene, M. Pepe, and M. Prelle, “MultiCore Framework: An API for Programming Heterogeneous Multicore Processors”, in *Proc. Of First Workshop on Software Tools for Multi-Core Systems*, Columbia University, NY, 2006.