Computing Spectropolarimetric Signals on Accelerator Hardware
Comparing the Cell BE and NVIDIA GPUs

Jose Garcia\textsuperscript{1}, Rory Kelly\textsuperscript{1}, Theron Voran\textsuperscript{1,2}
\{jgarcia, rory\}@ucar.edu, theron.voran@colorado.edu

\textsuperscript{1} National Center for Atmospheric Research (NCAR), Computational & Information
System Laboratory (CISL). P.O. Box 3000, Boulder, CO. 80307
\textsuperscript{2} University of Colorado, Department of Computer Science, 430 UCB, Boulder, CO. 80309

\textbf{Abstract.} Rapid calculation of the Voigt profile is critical for high performance in computational spectropolarimetric analysis. The Curtis and Osborne approximation to the Voigt function is arithmetically dense and embarrassingly parallel which makes it an intriguing candidate for exploiting accelerator technologies. We implement versions for the Cell Broadband Engine and for an NVIDIA GPU, and compare both the performance and the programmability of the two platforms.

1 Introduction

As single threaded performance gains in conventional microprocessors have slowed in recent years, the trend toward achieving application speedup through increased parallelism at the socket level has been gaining momentum. Standard CPUs have gone from single core variants to versions with two, four, six, and soon eight cores. On the far end of this parallelism trend are specialized accelerators such as the Cell Broadband Engine, with nine heterogeneous vector cores, and graphics cards like those from NVIDIA with hundreds of stream processing cores.

For applications to benefit from this trend they must be rewritten to take advantage of the available parallelism. This creates a software problem for developers who are often faced with not only a rewrite, but a restructuring of their code to a form that maps efficiently to the target architecture. This task can be expensive and laborious. Accelerator manufacturers offer software tools and programming environments aimed at simplifying the process of writing and optimizing code for their platforms. IBM offers a complete Cell Software Development Kit \cite{ibm} for the Cell BE which includes compilers, tools, standard libraries, and code examples. NVIDIA offers a novel programming language, the Compute Unified Device Architecture or CUDA, along with libraries, a compiler, tools, and source code examples \cite{nvidia} for programming their graphics cards. Additionally many standard mathematical libraries such as BLAS, Linpack, etc. are available for these architectures, from OEMs, third party vendors, and the open source
community. However, even with the increasing quality of programming tools, and growing number of libraries available, the process of porting a large application to an accelerator is highly complex and time consuming. Because of this, codes that are good initial candidates for accelerators should follow some sort of 90/10 rule, where a large fraction of the computation (i.e. 90%) occurs in a small fraction of the source code (i.e. 10%). There are many examples of this type of effort, such as the Weather Research and Forecast model Single Moment Tracer (WSM5) \cite{3} and the Wilson-Dirac Operator from Quantum Chromodynamics \cite{4}. In this paper we focus on another example, the Voigt function, which is a computational kernel taken from a spectropolarimetric signal analysis model.

2 Spectral Line Broadening and the Voigt Function

A spectral line is a bright or dark line in the frequency spectrum resulting from photon emission or absorption at a certain energy level. Spectral lines will experience broadening, i.e. the line will extend over several frequencies, which can have several causes:

- Natural broadening, described by a Cauchy-Lorentz distribution.
- Resonance broadening, described by a Cauchy-Lorentz distribution.
- Collision broadening, described by a Cauchy-Lorentz distribution.
- Thermal Doppler broadening, described by a Gaussian distribution.

All the broadening caused by these mechanisms can be denoted mathematically by the convolution of a Lorentzian and Gaussian distribution:

\[
(L * G)(f) = \int_{-\infty}^{\infty} L(\tau)G(f - \tau)d\tau .
\]

The resulting frequency dependent profile is called the Voigt profile\cite{5}, and is given by the normalized distribution \(P(x, y)\):

\[
P(x, y) = \frac{1}{\alpha_D} \left( \frac{\ln(2)}{\pi} \right)^{\frac{1}{2}} K(x, y) ,
\]

where \(x\) is a dimensionless frequency offset, \(y\) is damping parameter (the ratio of Lorentz to Doppler half-widths), and \(K(x, y)\) is the Voigt function, given by

\[
K(x, y) = \frac{y}{\pi} \int_{-\infty}^{\infty} \frac{e^{-t^2}}{y^2 + (x - t)^2} dt .
\]

Rapid calculation of the Voigt function is critical to the performance of computational spectropolarimetric models, where it can take up to 75% of the compute time \cite{6}\cite{7}. Most algorithms for computing the Voigt function use either the integral representation given by equation (3), or are based on the observation that the Voigt function is the real part of the complex error function\cite{8}. This approach tends to yield the best performance, and is the most common in
practice. The complex error function can, in turn, be written in terms of the complimentary error function, \( \text{erfc}(z) \) [9] as:

\[
w(z) = e^{-z^2} \text{erfc}(-iz),
\]  

(4)

Thus if we can find an efficient means of computing \( \text{erfc}(z) \) we can also efficiently compute the Voigt function. Curtis and Osborne presented a rational polynomial approximation to the complementary error function [10], given by:

\[
\text{erfc}(x) \approx R(x) = \sum_{i=0}^{p} a_i x^i + \sum_{i=0}^{p} b_i x^i.
\]

(5)

Our implementation comes from the Milne-Eddington gRid Linear Inversion Network (MERLIN) model[11], which uses this approximation for the \( p = 6 \) case. The coefficients for \( p = 6 \) can be obtained from Hui, et al.[9].

3 Test Cases and Reference CPU Implementations

In order to compare the performance of the Voigt function on different architectures we created a standard test application and several test cases. The test application is simply a driver that repeatedly calls a reference implementation of the Voigt function taken from the MERLIN [11] application. The MERLIN implementation is a C++ macro function, which takes two floating point values as input, an offset and a damping value, and produces two floating point outputs, the Voigt value and the Faraday value. Each function call contains 96 Flop, requires two words as input, produces two words as output, and can be computed in single precision, which gives it a computational intensity of 6 Flop/byte of data movement [12].

To test the problem with different amounts of input data we standardized on three test case sizes: small, medium, and large, which use two input and two output arrays of 4096\(^2\), 8192\(^2\), and 16384\(^2\) respectively. The small test case performs \( 1.61 \times 10^9 \) Flop, the medium test case performs \( 6.44 \times 10^9 \) Flop, and the large test case performs \( 25.8 \times 10^9 \) Flop. Testing a range of data sizes is necessary because the amount of memory required by the problem can determine what type of implementation can be used. For example the small and medium cases are able to fit completely in GPU memory, however the memory requirements of the large case are such that the implementation is forced to break the problem into smaller blocks and perform multiple transfers from CPU to GPU memory.

The original implementation of the Voigt function taken from MERLIN was a serial macro function. In order to have a good CPU baseline to compare accelerator performance against, we wrote optimized, multi-threaded and vectorized versions of the Voigt function for modern IBM and Intel microprocessors. The performance of the optimized CPU versions is shown in figure [1].

The threaded and vectorized implementations provide much better performance than the original: an increase from 1.77 GFlop/s to 9.68 GFlop/s, or an factor of 5.5x for the Intel Core2 Duo, and an increase from 2.11 GFlop/s to 5.87
Fig. 1. CPU performance for the medium test case.

GFlop/s, or a factor of 2.8x for the IBM POWER6. The performance increase is larger on the Intel machine because SSE provides a potential 4x improvement over scalar x86 floating point hardware, whereas VMX on the POWER6 only offer a potential 2x improvement. For both CPUs, performance is relatively independent of problem size as long as the problem fits in memory.

4 Accelerator Architectures
4.1 Architecture of The Cell BE
The Cell is a heterogeneous multi-core processor consisting of one Power Processing Element (PPE) and eight vector processors known as Synergistic Processing Elements (SPEs). The PPE is a two-way multithreaded PowerPC core with a 128-bit vector unit and two levels of cache. The PPE is mainly intended to run the OS and coordinate work among the SPEs. The SPEs consist of a 128-bit SIMD unit called the Synergistic Processing Unit (SPU), a 256 KB unified (instruction / data) memory known as the Local Store (LS) and a Memory Flow Controller (MFC) that manages DMA transfers to and from main memory. Each SPE has a theoretical peak of 25.6 GFlop/s, or 204.8 GFlop/s for all the SPEs together. The PPE and SPEs access memory and communicate with each other over the Element Interconnect Bus (EIB), a four lane, high bandwidth ring bus, capable of transferring data at 204.8 GB/s. The system used for this experiment was an IBM QS/22, containing two Cell processors (PowerXCell 8i) and 16 GB of 800MHz DDR2 memory.
4.2 Architecture of the NVIDIA 9800 GX2

The 9800 GX2 GPU is a massively parallel architecture containing three distinct levels of parallelism. On the coarsest level, the card contains two G92 GPUs able to communicate with the host system through a shared PCI-E x16 slot. Each G92 is a multicore chip containing 16 Streaming Multiprocessors (SMs); each SM contains 8 Scalar Processor (SP) cores, 2 transcendental function units, a Single Instruction Multiple Thread (SIMT) control unit and chip-local shared memory\cite{2}. The SIMT unit within each of the SMs is responsible for scheduling independent threads on its SP cores, each of which can execute multiple threads with independent register states. Each G92 chip has access to 512 MB of local DRAM and 64 KB constant memory, all of which can be accessed from any thread, on any SP within any of the SMs. Each SM also has access to 16 KB of fast local memory, 8192 32-bit registers and small read-only caches for constant and texture memory, all shared between the SP cores in the SM.

The 256 cores on the 9800 GX2 operate at 1.5 GHz, and each core can execute a scalar multiply-add per clock cycle, for a peak performance of 774 GFlop/s. Performance is somewhat higher if you allow for the 64 transcendental units. The DRAM in each of the G92 chips operates at 1.0 GHz, and transfers at double data rate over a 256-bit bus, giving each G92 64 GB/s of available memory bandwidth, or 128 MB/s aggregate for the entire card. The card communicates with the host system over a single PCI Express x16 connection, and supports both PCI Express 1.0, with 4 GB/s of bandwidth, or PCI Express 2.0, with 8 GB/s of bandwidth. Both G92 GPUs share a single PCI Express connection. Bandwidth between a G92 and its local DRAM is 32x greater than bandwidth to the host system, so transferring data to and from the GPU is a common bottleneck.

5 Accelerator Implementations

5.1 Cell BE Implementation

Running the SPEs, the pthreads model The Cell BE has nine programming models: Function-Offload, Device-Extension, Computation-Acceleration, Shared-Memory, Asymmetric-Thread Runtime, User-Mode Threads, Cell Application Frameworks, and SPE Overlays\cite{13}.

In this paper we focused on using the Asymmetric-Thread Runtime model since it fits the best with the numerical experiment as explained in section 3. In this model the PPE acts as a master and the SPEs act as slaves. Two independent programs must be written by the developer to fully implement the Voigt function experiment. The first program executes on the PPE. The objective of the PPE program is to coordinate the work among the multiple SPEs, to provide input data to each SPE and to receive and store output data from each SPE. The software model used to accomplish this task is the pthreads model. The main thread of execution in the PPE program creates a series of secondary threads and each one of these threads must communicate with an SPE, indicating when
input data is available for DMA transfers, wait for the signal from the SPE to indicate that SPE execution has completed, do a DMA transfer to collect the results and repeat the cycle with another block of data. The secondary threads must coordinate with the primary thread to partition the input data structure in order to asynchronously scatter the input and gather the results.

The program that executes on the SPEs takes the input data it is given and computes the value of the Voigt function. The numerical aspect of this program is simple, as it is just the implementation of the complex rational polynomial defined by equation (5), and is similar to the original CPU version, except that it operates on vectors. Most of the effort in this program is in making sure that the DMA transfers are efficient, and achieving effective data parallelism by vectorizing the computation on the SPEs.

**DMA Transfers and Multi-buffering** Each SPE relies on DMA transfers to move input data from the main memory to its Local Store (LS). In this experiment the total amount of data to be utilized by each SPE is much bigger than what can be stored in the LS at any given time. To move data in and out of the LS the following scheme is necessary:

1. Start a DMA transfer of input data from main memory to a buffer A in LS.
2. Wait for the transfer to complete.
3. Compute the Voigt function in buffer A (overwrite).
4. Initiate the transfer of the output Buffer A to main memory.
5. Wait for the transfer to complete.
6. Repeat.

It is evident time is being wasted in steps 2 and 5 because we are blocking the execution of the SPE waiting for the DMA to complete. We can improve the code if we overlap the DMA transfer with the computation of the Voigt function. This technique is known as multi-buffering. To achieve this, consider the modified scheme illustrated in figure 2. By constantly overlapping execution of the vectorized Voigt function with the DMA transfers, it is possible to get the naturally blocking `wait` calls to return immediately, hence reducing the wasted time.

![Fig. 2. DMA multibuffering scheme allows for the overlapping of DMA transfers with computation.](image-url)
Vectorizing on the SPEs and using the SPE intrinsic library. Given that the SPE contains a vector processor, exploiting parallel execution on vector data structures is paramount for achieving the maximum performance from the Cell BE. As stated in [4.1], the key for vector execution is the large register file which allows the SPU to store multiple values of the same data type and process them in parallel using the same operation with a vector instruction.

To expose vector operations to the developer, the Cell SDK offers a rich collection of C/C++ language extensions and functions called SPU intrinsics, which are essentially in-line assembly-language instructions written as C functions. The SPU intrinsic functions come in several types:

- Specific; have a one-to-one mapping with a single assembly instruction.
- Generic; map one or more Assembly instruction as a function.
- Composite; Convenience intrinsics composed as a sequence of specific or generic intrinsics.

Because the SPU intrinsics work at a very low level they do not always allow for easy development. In order to offer an easier way to use the vector capabilities of the SPU, the Cell SDK offers a set of vector intrinsics that are built around the C/C++ extension keyword `vector`. Once a data structure is defined as a `vector`, the essential operations of addition, subtraction, multiplication, insertion, etc. can be performed by the SPU in streaming mode with the vector intrinsic family of functions. Rewriting the Voigt function to use the vector intrinsics on the SPEs was essential in making the code run efficiently, and resulted in a large speedup versus the non-vectorized version.

5.2 NVIDIA GPU Implementation

The CUDA Programming Model. Applications written in CUDA are heterogeneous, with a portion of the application, including the main entry point, executing on a traditional microprocessor system, the host, and with kernel functions executing on the GPU or device. Kernel functions are executed concurrently by many CUDA threads. Threads are arranged into groups called thread blocks and blocks are arranged in a grid. Thread blocks can be one-, two- or three-dimensional; a thread’s index within its thread block is assigned through built-in index variables. Similarly, the grid can consist of a one-, two-, or three-dimensional collection of thread blocks, and the blocks are indexed through a second set of built-in indices. This allows flexibility in decomposing application data structures to map efficiently to the GPU hardware. The host and device both have their own DRAM memory. Device local DRAM is referred to as global memory; it is allocated and data is transferred to it, through calls to the CUDA run-time library. All threads in the grid have access to global device memory. Additionally all the threads in a block can access a common shared memory space which is significantly faster than global memory. Threads cannot access the shared memory of another thread block; threads from different blocks can only share data by writing it back to global memory. Finally each thread has
access to registers which can be used to store small amounts of data such as temporaries and intermediate results.

**Problem Decomposition** Decomposing the problem domain for the GPU must satisfy a few requirements. First, the domain must be broken into pieces that fit within the 512 MB global memories. Second, the decomposition should allow threads accessing global memory to take advantage of *coalesced* reads and writes. Finally, the decomposition should permit overlapping computation with PCIe data transfers by using multiple asynchronous *streams*. The domain for the test cases is 2-D, with axes representing frequency offset values and damping values. The Voigt function is computed for each offset-damping input pair in this domain. The host system allocates space to store input and output values as 1-D arrays. There are multiple GPUs in the host system, and each is managed by a separate CPU thread. Host data arrays are split into equally sized blocks of contiguous memory, with one block per CPU thread. Each CPU thread breaks its local block of data into contiguous chunks that fit into GPU memory, and each chunk is further divided into equally sized streams that execute asynchronously. Finally, the data within each stream is arranged into a 2-D grid of CUDA thread blocks, where each block is a 2-D arrangement of CUDA threads. Within each thread block the number of threads in $x$ and $y$ can be varied, and the total number of thread blocks is adjusted to maintain a one-to-one mapping of CUDA threads to points in the global problem domain.

**Data Transfer and Memory Access** Data transfer rates are a major constraint for this problem on the GPU. Although the 9800 GX2 has 128 GB/s of internal memory bandwidth, data transferred between the CPU and GPU must use the PCIe bus, which limits bandwidth to 4 GB/s (PCIe 1.0) or 8 GB/s (PCIe 2.0). The host system used in this paper only supported PCIe 1.0, and host to device DMA bandwidth was the largest performance bottleneck for this application. DMA transfers are initiated on the host; the CUDA runtime copies data from arrays in host memory into page-locked buffers before the DMA can occur. The copy can be avoided by directly allocating host arrays in pinned memory, allowing DMAs to achieve maximal effective bandwidth from the PCIe bus. Using pinned memory led to a near doubling of performance versus pageable memory. Another strategy for increasing the effective bandwidth is to use multiple asynchronous streams. This allows overlapping of the DMAs with kernel computation. When using more than four streams the benefit of overlapping DMAs with computation was balanced by the increased management overhead and lower bandwidth for smaller DMAs. For most cases two streams was optimal and resulted in a 10% to 15% gain over a single stream. Once data is in device memory, it is critical to ensure efficient access by the CUDA threads. To achieve maximal bandwidth from device memory, accesses must be *coalesced*, i.e. the memory access pattern is such that multiple memory transactions within a half-warp (16 threads) are combined into a single coordinated memory access. Properly coalesced memory access can achieve bandwidths an order of magni-
tude higher than non-coalesced transactions. Coalescing on the 9800 GX2 requires data accessed to be in a contiguous region of memory, and each thread within the half-warp to access a single word of 4, 8, or 16 bytes. Additionally the threads must access the words in sequence - the \( k^{th} \) thread in the half-warp must access the \( k^{th} \) word in the memory segment. All threads in a block also have access to local shared memory and a pool of registers, both of which are faster than global memory. To limit accesses to global memory, the input data needed by each thread was copied into a register at the beginning of the kernel, intermediate calculations were performed in registers, and output data was copied back to global memory at the end of the kernel. Memory coalescing and limiting read and writes to global memory, ensured that local memory was not a performance bottleneck, and allowed efficient kernel execution.

**Optimizing the Computation** The cost of the computation is only a small percentage of total execution time, which is dominated by host-device data transfer. The kernel computation itself is simple and required few changes from the scalar CPU code. Only a few optimizations were made to ensure that the computation executed as efficiently as possible. Replacing floating point division with a fast intrinsic and reordering calculations to increase the use of FMAs resulted in a modest kernel speedup and did not noticeably impact the accuracy of the overall calculation.

### 6 Performance and Results

The Cell BE and NVIDIA GPU versions of the Voigt function were run multiple times to find optimal configurations. Once the optimal configurations were found multiple runs were performed to measure performance of the accelerated functions. Table I lists the performance of the algorithms for all three test cases, along with the time spent computing and the time spent transferring data to the accelerators. As can be seen from the table data transfer overhead is a bottleneck on both the Cell BE and the NVIDIA GPUs. Figure 3 shows the performance of both accelerators compared to the highly optimized CPU version. Specifics for both accelerators are discussed below.

#### 6.1 Cell BE

On the Cell BE, for each experiment, the PPE simply starts the SPEs and the SPEs run the DMA engine as a conveyor belt moving in points of the Voigt function domain to be computed and moving the results back out to main memory. Results for the different test case sizes demonstrate how the overhead of starting the computation gets amortized away for large inputs. As can be seen in table I the data transfer overhead is a significant bottleneck for the Cell BE. The time it takes to transfer data to and from the SPEs accounts for more than half of the total time it takes to execute the algorithm. This indicates that the problem does not have a high enough degree of computational intensity to make full use
Fig. 3. Comparison of Cell BE and NVIDIA GPU versions of Voigt function with the optimized CPU version (Intel Core2 Duo with SSE).

Table 1.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Performance Results</th>
<th>Cell BE</th>
<th>NVIDIA GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>Total Performance (GFlop/s)</td>
<td>34.17</td>
<td>30.78</td>
</tr>
<tr>
<td></td>
<td>Total Time (ms)</td>
<td>47.13</td>
<td>52.33</td>
</tr>
<tr>
<td></td>
<td>Compute Time (ms)</td>
<td>17.56</td>
<td>3.43</td>
</tr>
<tr>
<td></td>
<td>Transfer Time (ms)</td>
<td>29.57</td>
<td>48.90</td>
</tr>
<tr>
<td></td>
<td>Ratio Transfer/Total (%)</td>
<td>62.74</td>
<td>93.44</td>
</tr>
<tr>
<td>Medium</td>
<td>Total Performance (GFlop/s)</td>
<td>47.45</td>
<td>30.93</td>
</tr>
<tr>
<td></td>
<td>Total Time (ms)</td>
<td>135.78</td>
<td>208.29</td>
</tr>
<tr>
<td></td>
<td>Compute Time (ms)</td>
<td>63.31</td>
<td>13.83</td>
</tr>
<tr>
<td></td>
<td>Transfer Time (ms)</td>
<td>72.47</td>
<td>194.46</td>
</tr>
<tr>
<td></td>
<td>Ratio Transfer/Total (%)</td>
<td>53.37</td>
<td>93.36</td>
</tr>
<tr>
<td>Large</td>
<td>Total Performance (GFlop/s)</td>
<td>52.33</td>
<td>28.63</td>
</tr>
<tr>
<td></td>
<td>Total Time (ms)</td>
<td>492.45</td>
<td>900.10</td>
</tr>
<tr>
<td></td>
<td>Compute Time (ms)</td>
<td>223.29</td>
<td>54.61</td>
</tr>
<tr>
<td></td>
<td>Transfer Time (ms)</td>
<td>269.16</td>
<td>845.49</td>
</tr>
<tr>
<td></td>
<td>Ratio Transfer/Total (%)</td>
<td>54.66</td>
<td>93.93</td>
</tr>
</tbody>
</table>
Computing Spectropolarimetric Signals on Accelerator Hardware 11

of the floating point power available on the Cell. Even though this is the case, the Cell BE is still able to get a speedup of over 5x compared to the optimized CPU version of the code, or a speedup of about 25x vs the original serial code.

6.2 NVIDIA GPU

As can be seen in table 1, the maximum performance achieved by the CUDA version of the Voigt function is 30.93 GFlop/s, which represents only 2.0% of the 1548 GFlop/s peak capacity of the system. This poor performance is due to two primary causes: the limited bandwidth of the PCI Express 1.0 bus, and the performance of the host memory sub-system. The PCIe bottleneck is by far the most critical factor. For typical cases, around of 93% of the total run time is spent transferring data over the PCIe bus. The host system considered here supports only PCIe 1.0 data rates of 4.0 GB/s. The PCIe bandwidth is such a major bottleneck for this problem that a reasonable model of GPU performance is simply to assume that computing is free, and that computations can proceed as quickly as the PCIe bus can transfer data. With peak PCIe bandwidth of 4.0 GB/s, and taking into account protocol overhead, one should expect to get in the range of 3.5 to 3.7 GB/s from the PCIe bus. Assuming we can get 3.6 GB/s, and assuming that computation is free, then since each Voigt function invocation performs 96 Flops, and requires transferring 4 4-Byte values (6 Flop/byte), we should be able to compute at a rate of 6 Flop/Byte × 3.6 GB/s = 21.6 GFlop/s.
quite close to the values we actually see on a single card, as seen in figure 4.
Upgrading to a host system with PCIe 2.0 doubles PCIe bandwidth and should
give close to a factor of 2x in overall performance. However, the memory system
needs to be able to feed the faster PCIe system. In the system tested here, the
memory system can only sustain 5700 MB/s of memory bandwidth, which is
not sufficient to saturate both cards operating simultaneously, even over PCIe
1.0. This is why performance does not double when using two cards - in the two
card case the bandwidth of the main memory system becomes the performance
bottleneck. Despite these limitations the NVIDIA 9800 GX2 system is still able
to achieve a speedup of roughly 3.5x over the optimized CPU version of the code,
or about 19x over the original serial code.

6.3 Artificially Increasing Computational Intensity

It is clear from the sections 6.1 and 6.2 that neither of the accelerators is able to
achieve anywhere near their peak performances on this problem, due to the fact
that the computational intensity of the Voigt function is too low. With only 6
Flop/byte to work with, the Cell BE system is only able to achieve a max of 53
GFlop/s or 13% of the 409.6 GFlop/s peak for the 16 SPEs in the system. The
NVIDIA system is only able to achieve 30 GFlop/s, or 2% of the 1548 GFlop/s
peak for the two 9800 GX2 cards in the system. It is natural to ask how these
systems would perform on codes that are similarly structured, i.e. arithmetically
dense, but with more Flops performed for each byte of memory transferred. To
answer this question we created an artificial test case which performs multiple
calculations of the Voigt function for each input / output pair, and used it to
generate curves of performance vs. computational intensity, which are shown in
figure 5.

Both systems achieve a much larger percentage of their peak performance as
the computational intensity of the problem increases. The performance of the
Cell BE system increases steadily until it begins to level off at a computational
intensity of around 150 Flop/byte, where it is able to achieve 128 GFlop/s, or
about 32% of peak. Performance on the NVIDIA system continues to increase
until around 3000 Flop/byte, where it is sustaining roughly 1.2 TFlop/s, or
about 78% of peak performance.

7 Comparisons, Impressions, and Conclusions

We had several goals in implementing the Voigt function for these different
accelerators: to compare performance on our test problem, to learn about the
programming process and the effort required to write code for both platforms,
and to gain an understanding of their performance characteristics and how we
might envision using them within a more realistic model as opposed to a simple
driver routine for a small kernel.

Apples-to-apples comparisons of performance are problematic because of the
differences in the platforms themselves. The Cell BE system, where the PPE and
As data transfer overhead is amortized away, the accelerator systems are able to achieve a larger fraction of their peak performance. The SPEs are integrated at the die level and share common memory, is a very different hardware setup from the CPU-GPU model, where the two different processor must communicate over a PCIe bus. For purposes of comparing the architectures as accelerators, it may be better to consider the GPU in isolation from the host system and only compare performance within the GPU to the Cell BE. Or it may make more sense to compare the CPU-GPU system to a system where a QS-22 blade is attached to a traditional CPU node through a PCIe bus, as is the case in machines such as the Roadrunner supercomputer at Los Alamos. When comparing the GPU in isolation to the Cell BE, and taking, as an example, the medium test case, the GPU machine spends about 194 msec transferring data over the PCIe bus, and 13.8 msec to read from local memory, compute the kernel, and write the results back to memory. This translates into a GPU-only computation rate of roughly 467 GFlop/s, vs. roughly 47 GFlop/s for the Cell BE. If instead you imagine the QS-22 as an accelerator connected to a traditional CPU host system, then performing the equivalent communication over the PCIe bus should add around 194 msec to the execution time, which would degrade performance to roughly 20 GFlop/s in a hypothetical roadrunner type configuration, compared to 30 GFlop/s for the CPU-GPU system. However, these comparisons are unfair to the Cell BE because they ignore that the Cell has capabilities a GPU in isolation simply does not posses. Using a GPU as an accelerator requires attaching it to a CPU host, whereas the Cell BE is...
fully capable of hosting itself, running an operating system, performing network communication and file I/O etc. This ability allows for systems such a standalone QS-22, or clusters of purely Cell based nodes. If the goal was to build a machine with the aim of repeatedly calculating the Voigt function as rapidly as possible, then a QS-22 is a better solution. By themselves these results are relevant only to the specifics of our experiment but the conclusion to be drawn from them is that the peak performance of an accelerator is only the most basic indicator of its capacity. The bandwidth available to the accelerator, the details of the host hardware, the details of the larger system node architecture, how nodes are interconnected, the types of the problems to be solved and so on, are all critical details that must be considered before designing an accelerator based system.

Another goal of this research was to get insight into the level of involvement required for a programmer to write effective code for these platforms. In case of the Cell BE we conclude that for a well rounded Unix programmer with experience in interprocess communication, POSIX threads, vectorization, etc. the Cell SDK is not a difficult programming environment to learn. Much of the Cell SDK is similar to common programming practices in the Unix (and particularly the HPC) world. This is an important consideration for productivity in the Cell BE environment as individuals for which these concepts are new may not initially be as effective as an experienced Unix programmer. However, for the experienced programmer the learning curve is not too steep and she can expect to become productive in the Cell programming model within a short period of time. The CUDA programming environment is, we feel, conceptually simpler than programming for the Cell BE. Most competent programmers, even those without much prior experience in parallel computing, should be able to learn CUDA programming relatively quickly. The main conceptual shift is from a style of computing where computations occur on data elements by indices computed in loops, to a paradigm where indices are mapped to individual threads which execute concurrently. Learning CUDA might be compared in difficulty to learning parallel programming with OpenMP. Both IBM and NVIDIA provide a wealth of free tools, documentation, tutorials, and example code to help new programmers get started, and in both cases the quality of the available materials is excellent.

We have learned a great deal about how accelerators can be exploited for a simple data parallel algorithm. We have seen that the key ingredient dominating performance is the cost associated with data movement to the computational units on the accelerators, and not the computational cost itself. This remains one of the main challenges of the accelerator model, that is, what types of algorithms are well suited to the architecture of a given machine and what degree of computational intensity is required before the cost of moving data to an accelerator is outweighed by the speedup gained from using it. In the future we plan to continue exploring this question, both for problems with data dependencies requiring increased communication and synchronization, and for larger clustered machine setups containing accelerator nodes.
8 Source code

The code for the numerical experiments presented in this paper can be obtained at [http://www.cisl.ucar.edu/css/software/](http://www.cisl.ucar.edu/css/software/) or by writing directly to any of the authors.

References