Implementation of 2-D FFT on the Cell Broadband Engine Architecture

Kerry Barnes (kbarnes@gedae.com), William Lundgren (wlundgren@gedae.com), James Steed (jsteed@gedae.com)
Gedae, Inc., 1247 N. Church St., Suite 5, Moorestown, NJ 08057

Introduction

The bandwidth between local fast memory and remote slow memory is a major roadblock in achieving high performance in software from modern multicore hardware. Careful data flow planning and organization must be done to make sure data arrives just as it is ready to be processed, overlapping communication and processing to maintain throughput. The Cell Broadband Engine (Cell/B.E.) processor provides a template for these programming concerns. Many other architectures, from cluster computing with Intel or IBM Power processors to the upcoming Intel Larrabee GPGPU (general processing GPU), create similar challenges to the programmer. To illustrate the data flow planning required to achieve near optimal performance on these modern hierarchical memory architectures, we investigate the implementation of a 2-D Fast Fourier Transform (FFT) on the Cell/B.E. processor. The Gedae language and compiler is used to quickly translate data flow strategies into efficient Cell/B.E. implementations.

Cell Broadband Engine and its Transfer Limitations

The Cell/B.E. processor has 8 identical Synergistic Processing Elements (SPE) alongside a Power Processing Element (PPE). Each SPE has its own 256 kB local store (LS) and DMA engine as shown in Figure 1. Larger system memory is available via a memory controller with a bandwidth of 25.6 GB/s – much less than the high speed Element Interconnect Bus (EIB). Each SPE is capable of processing 25.6 GFLOPS when running at 3.2 GHz for an aggregate speed of 204.8 GFLOPS. When processing large data sets, data is often stripmined from the system memory. In these situations the bandwidth over the memory controller quickly becomes a bottleneck to the processing speed.

While system memory is outside of the SPEs' memory space, the SPE can transfer data from system memory using a direct memory access (DMA) API to put or get memory in the unmapped address space. The Cell Software Development Kit (Cell SDK) supports both vanilla and list DMA transfers. Vanilla DMA transfers transfer contiguous off-chip memory to contiguous on-chip memory. List DMA transfers transfer tiles to/from system memory from/to contiguous on-chip memory where a tile is defined as a submatrix of a larger data set with N contiguous elements per row and M elements between rows such that M>N.

To efficiently use these DMA transfers, several design considerations must be taken into account. There is a boundary size of 16 bytes on all transfers. The EIB has a width of 128 bytes, and it follows that the contiguous data (including elements in a tile's row) should be at least 128 bytes. Additional experimentation was done to stress the memory controller bandwidth and determine the optimal transfer size, and the row length of 2048 bytes was found to be best, offering 4x improvement over 128 byte row size, as shown in Figure 2.

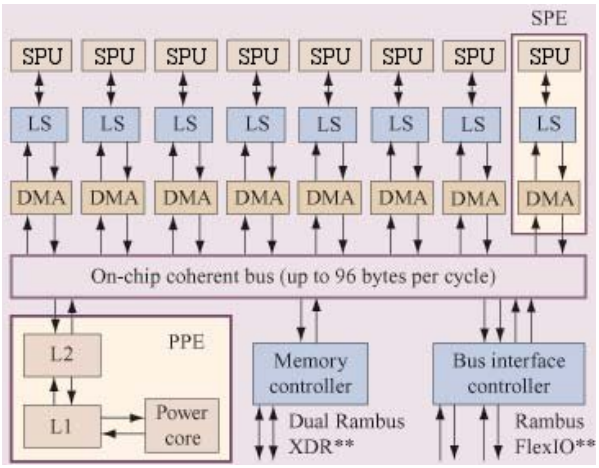

Figure 1 – The current Cell/B.E. processor combines the PPE with 8 SPEs. The memory controller often is a bottleneck.

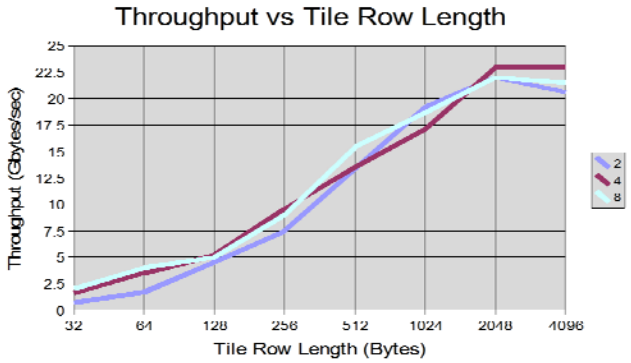

Figure 2 – Maximum throughput over the memory controller is achieved when DMA list row sizes are at least 2048 bytes.

Algorithms

The 2-D FFT is implemented on the Cell Broadband Engine architecture. We consider the data size 512x512 and larger – the data is too large to fit in a single SPE's local storage, and it is too large to fit in the aggregate local storage. Data must be stripmined from system memory, including intermediate stages (transposes) in the algorithm. We present two algorithms that can accommodate these sizes – a three phase algorithm and a four phase algorithm. The three phase algorithm can accommodate larger matrices at the cost of more transfers to and from system memory.

Let P be the number of processors, R*C be the input matrix size, and the matrix be divided into tiles such that R=R1*R2 and C=C1*C2, i.e., R1*C1 tiles of size R2*C2.

The 3 phase algorithm is

- Perform the FFT on C/P stripmined columns of size R into and out of system memory
- Perform the transpose on R1*C1/P tiles of size R2*C2 into and out of system memory
- Perform the FFT on R/P stripmined columns of size C into and out of system memory.

The 3 phase algorithm requires 6 transfers to and from system memory – 4 vanilla DMA transfers and 2 list DMA transfers. The 4 phase algorithm can only be performed on matrices up to 512x512, but it reduces the number of transfers to and from system memory. The algorithm is

- Repeat C1 times
  - Perform the FFT on C2/p stripmined columns from system memory to local storage
  - Stripmine C2/P*R2/P tiles from the FFT result in local storage back to system memory
- Repeat R1 times
  - Stripmine R2/P*C2/P tiles out of system memory to local storage
  - Perform the FFT and stripmine the R2/P columns back into system memory

The 4 phase algorithm requires only 4 transfers to and from system memory – 2 vanilla DMA transfers and 2 list DMA transfers.

With both algorithms, the DMA work overlaps with the work of the transpose and the FFT through the use of double buffered reads and writes to system memory.

The goal of reduction in total number of transfers adds an additional design requirement when considering the complex data storage type. The most efficient kernel for performing an FFT on the SPE's ALU with VMX instruction set uses the split complex data type, i.e., the real and imaginary buffers are stored separately, not interleaved. Transferring two buffers to and from system memory adds overhead, so the buffers are allocated adjacently. This design allows bigger more efficient transfers with half of the DMA kick-off overhead. However, this consideration must be addressed when transposing the data.

Performance and Conclusions

The Gedae language and compiler are used to achieve high performance with minimal development time. The language allows for direct specification of the tiling of large matrices. The compiler automates a bare metal implementation of the DMA and DMA list accesses. The runtime components are autocoded to introduce zero overhead to the execution time of the algorithms. Additionally, Cell/B.E.-specific optimization strategies are automatically incorporated, such as the use of huge TLB pages to optimize memory-to-memory IPC and the use of optimized vector routines for the FFT and transpose operations.

The upper bound on performance is dependent on the transfer rate over the system memory controller. The theoretical bandwidth limit is 25.6 GB/s. As shown in Figure 2, the maximum we have achieved in the laboratory while simultaneously using all 8 SPEs is 22 GB/s. We use the equation $10*N^2*\log_2 N$ to compute the number of FLOPS for the 2-D FFT algorithm. Assuming a 25.6 GB/s bandwidth, the upper bound is 72 GFLOPS. Assuming a 22 GB/s bandwidth, the upper bound is 62 GFLOPS. For the current implementation we are measuring 51.5 GFLOPS for the four phase algorithm, which equates to an 18.3 GB/s sustained bandwidth over the memory controller. A Trace Table of this execution is shown in Figure 3. We have identified several additional optimizations that will be incorporated and announced at the conference to bring the performance close to the theoretical max.

Modern chips add more coprocessors with wider vector ALUs offering great promise of high throughput. As the chips provide more FLOPS, the programming challenge moves towards keeping the pipelines sated with data. Gedae provides a powerful platform for programming and debugging these bandwidth issues.

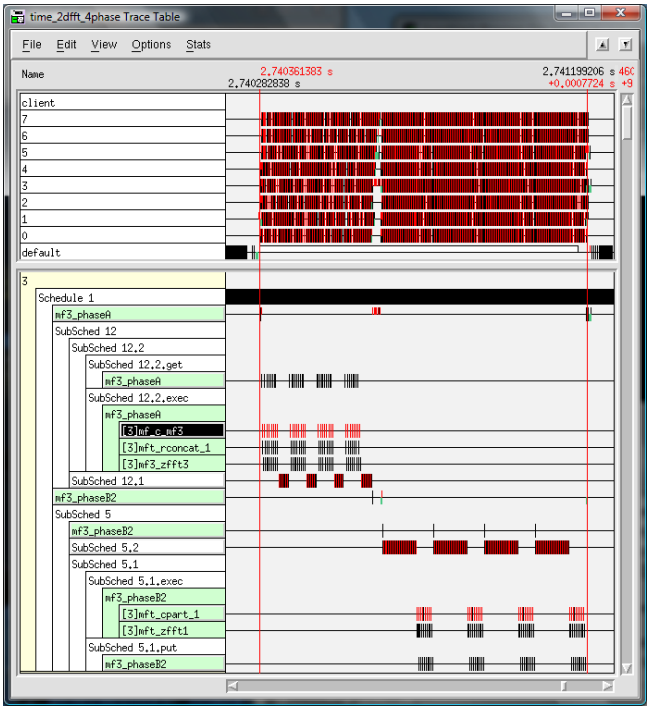

Figure 3 – Execution trace in the Gedae trace table shows the loading of the 8 processors and the staging of the 4 phase algorithm – get, row FFT, corner turn, column FFT, put.

References

[1] Greene, J. and R. Cooper. "A Parallel 64K Complex FFT Algorithm for the IBM/Sony/Toshiba Cell Broadband Engine Processor, " GSPx, 2005.

[2] IBM, Sony Computer Entertainment, Toshiba. *Cell Broadband Engine Programming Handbook*, Version 1.1, April 2007. <http://www.ibm.com>.

[3] Lundgren, W. et al. "Simple, Efficient, Portable Decomposition of Large Data Sets," HPEC, 2008.