Performance of a Multicore Matrix Multiplication Library

Frank Lauginiger (flauginiger@mc.com), Robert Cooper (rcooper@mc.com), Jonathan Greene (greene@mc.com), Michael Pepe (mpepe@mc.com), Myra Jean Prelle (mjp@mc.com)
Mercury Computer Systems, Inc., 199 Riverneck Road, Chelmsford, MA 01824

Abstract

Multicore processors promise dramatic improvements in performance, but their diverse and often unique architectures are a major inhibitor to software adoption. Algorithm libraries that operate at the chip level and are optimized across multiple cores provide the quickest route by which programmers can port or develop high-performance software for multicores. This paper reports on a flexible matrix multiplication library for the Cell Broadband Engine™ (BE) processor that meets or exceeds the performance of known matrix multiplication implementations on the Cell. In addition, the library operates within a larger framework for programming multicores that enables programmers to combine library code with multicore functions they have developed themselves.

Motivation

Multicore processors have appeared or been announced from many vendors. Programming has emerged as the major challenge to the adoption of multicore processors and the ability of applications to reach the promised performance gains.

Although parallel programming techniques have been developed over the last several decades, most programmers have yet to adopt them. As a consequence, existing programs and programmers are currently not well positioned to exploit multicores.

The most promising approach that offers both performance and ease of adoption is the function offload model. In this model, the main application thread running on a general-purpose core calls compute-intensive functions whose implementations offload work to multiple, possibly specialized, cores. By offering common algorithm libraries with optimized multicore implementations, we can bring a large group of programmers into the multicore realm, and provide software portability among multicore architectures.

The Cell Broadband Engine Architecture

The Cell BE [1] contains a single general-purpose Power™ architecture core and eight high-performance cores, termed Synergistic Processing Elements (SPEs). With the chip running at 3.2 GHz, each SPE has peak performance of 25.6 GFLOPS. The SPE does not have a conventional hardware cache; instead, each SPE has a 256KB local store with a bandwidth of 51.2 GB/s. This is the only memory to which the SPE has load/store access. To access main memory, the SPE must use DMA commands. The XDR DRAM main memory provides aggregate bandwidth of 25.6 GB/s.

In addition, the Cell provides very high SPE-to-SPE bandwidth using the Element Interconnect Bus (EIB). An SPE can communicate at 25.6 GB/s to any combination of other SPEs over the EIB. In the aggregate, the EIB provides 204.8 GB/s of bandwidth; thus in theory, every pair of SPEs could simultaneously communicate at 25.6 GB/s. Note that the general purpose core, external I/O and XDR memory also share the EIB. Exploiting explicit SPE-to-SPE communication is often the key to achieving the best performance on the Cell.

Problem Specification

For the matrix multiplication: $C = A * B$, we define the problem size with three parameters: the number of rows (nr) and columns (nc) of the result matrix, and the dot product length (dpl) of the row-by-column dot product that produces each element of the result matrix (Figure 1).

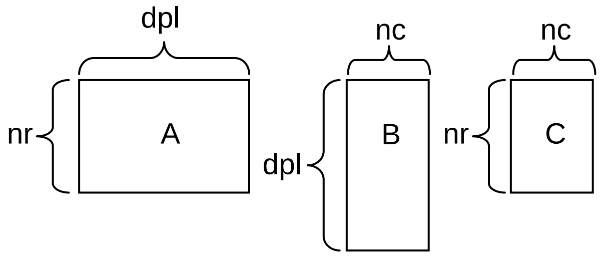

Figure 1: Matrix dimensions.

Thus C is of dimension nr rows by nc columns, A is of dimension nr rows by dpl columns and B is of dimension dpl rows by nc columns.

The library supports rectangular matrices in sizes with granularity of 32 rows or columns stored in row-major format. It also supports optional accumulation ($C = C + A * B$) and pre-transposition of either or both input matrices A and B. Finally, the user can select the number of SPEs (p) to use in the calculation.

Algorithm Mapping

The library uses different decompositions depending on the matrix size. Here we describe the algorithm mapping for matrices for which dpl is between 32 and 1024 elements.

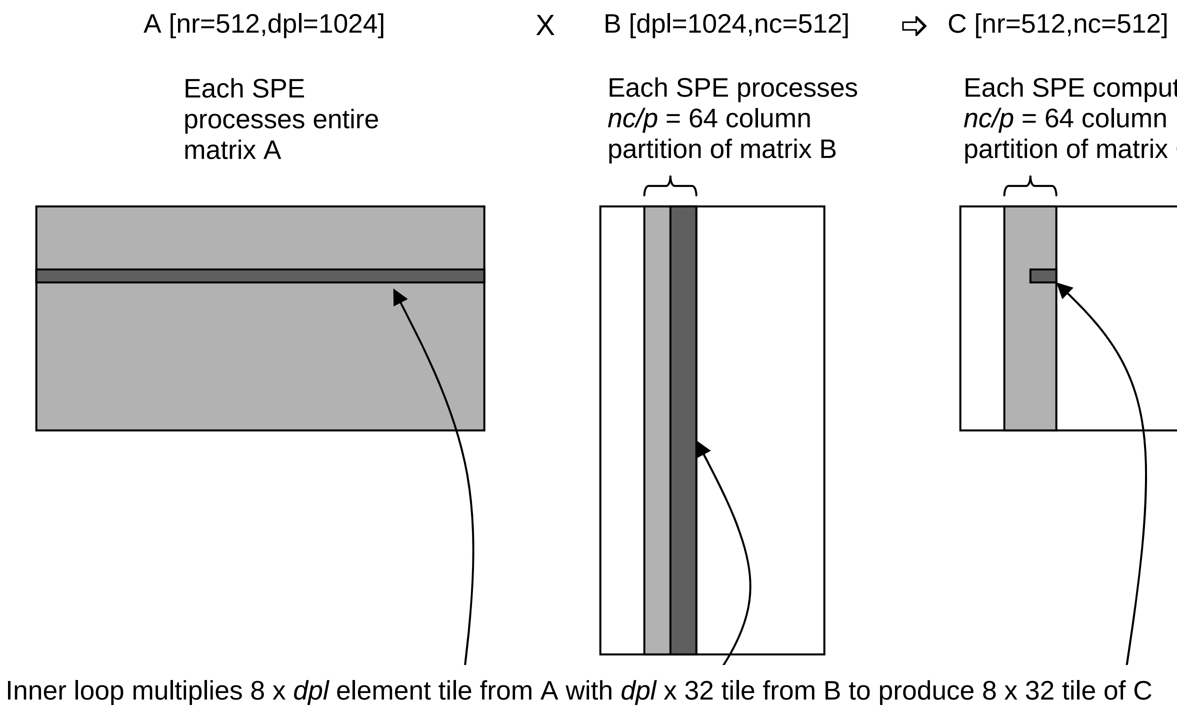
A [nr=512,dpl=1024]
X
B [dpl=1024,nc=512]
⇨
C [nr=512,nc=512]
Each SPE processes entire matrix A
Each SPE processes *nc/p* = 64 column partition of matrix B
Each SPE computes *nc/p* = 64 column partition of matrix C
Inner loop multiplies 8 x *dpl* element tile from A with *dpl* x 32 tile from B to produce 8 x 32 tile of C

Figure 2: Problem decomposition for multiplying a 512x1024 matrix by a 1024x512 matrix

Each SPE is responsible for producing an *nc/p* column partition of the result matrix (**Error! Reference source not found.**). To do so, it must read all rows of matrix *A* and an *nc/p* column partition from *B*. This partitioning makes each core's computation independent of all the others. However, each core will need to access the entire *A* matrix during the computation of its segment of the output matrix.

An SPE's partition of *B* is further subdivided into strips of 32 columns. For each column, the SPE proceeds to DMA all rows of matrix *A*, eight rows at a time from XDR memory. The inner loop then performs the dot products to produce an 8 x 32 tile in the output matrix *C*. For efficiency, this inner loop is written in highly tuned assembly language.

The choice of output tile size is critically determined by many competing tradeoffs including exploiting the full SIMD width, maximizing DMA size, and ensuring DMA alignment, all while not exceeding the 256KB local store size on an SPE.

The key to the performance of this algorithm is the retrieval of tiles from matrix *A*. This must be done by all SPEs for every 8x32 tile of *C*. If all eight SPEs independently read tiles, the XDR memory bandwidth will be a bottleneck.

A better option is to have a subset of SPEs each stream matrix *A* into its own local store, signal its neighbor(s), and then have these "neighbor" SPEs stream matrix *A* in over the much faster on-chip EIB. This idea is also used in [2].

We experimented with three DMA strategies, described here for the case of eight SPEs:

1. Even numbered SPEs stream in *A* from off-chip memory, signal the odd SPEs when the off-chip DMA is complete. Odd SPUs then stream in the *A* tile from its even neighbor over the EIB.
2. SPEs 0 and 4 pull the *A* tile in from external memory, all other SPEs pull from one of these two SPEs.
3. SPE 0 pulls from external memory; all other SPEs pull over the EIB from SPE 0.

The goal is to balance the double buffered DMA activity with the matrix multiplication's floating-point activity, which results in a virtual overlap of the two. We measured the total algorithm time for a 1024x1024 matrix multiply for each of these options on a 3.2 GHz Cell BE processor as shown in Table 1.

Table 1: DMA streaming performance

SPEs Which Pull A From XDR	GFLOPS
SPEs 0 through 7	159
SPEs 0, 2, 4 and 6	170
SPEs 0 and 4	169
SPE 0	170

The best case represents 83% efficiency compared to the eight SPE's collective peak performance of 204.8 GFLOPS.

Using the EIB to transfer tiles of matrix *A* means that each core will share a buffer with its neighbors. Therefore reading and writing to this buffer must be correctly and efficiently synchronized. This is done via DMA-based semaphores using the *dma_semaphore_[give,take]* routines of Mercury's MultiCore Framework [3].

Results

Preliminary performance results utilizing eight SPEs on a 3.2 GHz Cell BE processor are reported in Table 2.

Table 2: Matrix multiplication library performance

Matrix Dimensions nr	nc	dpl	GFLOPS	Efficiency
512	512	512	149	73%
512	512	1024	162	79%
768	768	768	163	79%
1024	1024	1024	170	83%

These results indicate that the performance increases as the dot product length gets larger. This is because the assembly code routine spends more time in the inner loop, making the overhead in the routine relatively smaller. The overhead is also a function of the DMA tile dimensions used. This could be reduced for smaller matrices by dynamically configuring the DMA tiling dimensions based on the overall matrix dimensions passed in at execution time.

Increasing the performance for smaller matrix dimensions, and extending the implementation to arbitrarily large matrices are the focus of our current development.

Related Work

Table 3 summarizes the performance of two other Cell matrix multiplication implementations. The Sony-Toshiba-IBM implementation is taken from the Cell BE Software Developer's Kit (SDK) [4]. It is limited to square, power-of-two sized matrices in block data layout rather than the row major layout used in our work. We ran the code from SDK 1.1 on a 3.2 GHz Cell processor with GNU toolchain 3.2, on FedoraCore 5 using 64KB pages. Our measurements differ from those reported by IBM [5], perhaps due to different toolchain versions.

Hackenberg's implementation [6] is limited to square matrices in size increments of 64, provides both row-major and block data layout versions, and supports accumulation.

Table 3: Related work performance (GFLOPS)

Matrix Dimension	STI SDK (block data layout)		Hackenberg (row major)
	Measured	Reported	Reported
512 x 512	138	201	70
768 x 768			125
1024 x 1024	167	201	150

While both these implementations report close to peak performance (200 GFLOPS) on larger matrices, our implementation achieves higher performance for matrix sizes on which we have focused.

Library and API Concerns

When developing a library, requirements beyond pure benchmark performance must be addressed. Because a given library function performs just part of the user's computation, it is important to provide flexible options for data layout, matrix size, and features such as accumulation. Without these options, the user's program may incur a performance penalty in order to accommodate the restrictions of the library function. Our implementation meets these requirements while maintaining high performance and a small-code footprint on the SPE.

Programming Multicores

An optimized offload library, such as described in this paper, is only one component of a high-performance multicore application. To achieve further optimizations, some custom algorithm work is recommended. For this reason, our library ensures that shrink-wrapped functions, such as described in this paper, can be used alongside explicitly programmed algorithms implemented with MultiCore Framework [3]. In particular, the task management, synchronization and resource management features of MCF are used cooperatively by the library.

Thus an effective development strategy for the Cell is to begin using the general-purpose Power core exclusively; replace local library function calls for compute-intensive operations with calls to an offload library; and, finally, to identify areas for further improvement that will benefit from custom algorithm development aided by multicore programming tools. They work described in this paper achieves that goal by combining maximum performance and flexibility in a single library.

References

[1] H. P. Hofstee, "Power Efficient Processor Architecture and The Cell Processor," *11th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2005.

[2] Y. Steinsaltz, S. Geaghan, M. J. Prelle, B. Bouzas, "Leveraging Multicomputer Frameworks for Use in Multi-Core Processors," *Tenth Annual High Performance Embedded Computing Conference (HPEC).* Sep. 21, 2006.

[3] B. Bouzas, R. Cooper, J. Greene, M. Pepe, M. J. Prelle, "MultiCore Framework: An API for Programming Heterogeneous Multicore Processors," *First Workshop on Software Tools for Multi-Core Systems (STMCS)*, March 26, 2006. http://www.isi.edu/~kintali/stmcs06/cooper.pdf

[4] Sony-Toshiba-IBM Design Center, *Cell BE Software Developers Kit (SDK), Version 1.1,* Dec. 2006. Available at: http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk1.1/.

[5] T. Chen, R. Raghavan, J. Dale, E. Iwata, *Cell Broadband Engine Architecture and its first implementation.* Nov. 2005. http://www.ibm.com/developerworks/power/library/pa-cellperf/.

[6] D. Hackenberg, "Performance Measurements on Cell SMP Systems," *Cell Cluster Meeting in Jülich*, May 10, 2007. http://www.fz-juelich.de/zam/datapool/cell/Performance_Measurements_on_Cell.pdf.