Evaluating Cell/B.E Software Cache for ClustalW

Vipin Sachdeva
IBM Systems and Technology Group
Indianapolis, IN
vsachde@us.ibm.com

Michael Kistler
IBM Austin Research Lab
Austin, TX
mkistler@us.ibm.com

David A. Bader
College of Computing
Georgia Institute of Technology
Atlanta, GA

Abstract— **This paper evaluates the performance of the bioinformatics application ClustalW developed on Cell Broadband Engine(TM) (Cell/B.E.) using a software data cache for SPEs, instead of explicit DMA transfers. The software cache of the SPEs, once it has been configured, provides the capability to access main memory with data-transfer functions that override the need for DMA commands. ClustalW exhibits high spatial locality but little temporal locality. We compare performance of ClustalW with a previous version that uses explicit DMA transfers as a means of communication with the system memory, and provide analysis and results of our comparison.**

I. Introduction

The recent emergence of accelerator technologies like GPUs, FPGAs and specialized processors has made it possible to achieve an order-of-magnitude improvement in execution time for many bioinformatics applications compared to current general-purpose platforms. Although these accelerator technologies have a performance advantage, they are constrained by the high effort needed in porting the application to these platforms. Recent implementations have focused on sequence similarity applications using a wide variety of accelerators including Cell/B.E. [1], GPUs [2] and FPGAs [3].

In previous research [1], we ported and optimized two key sequence similarity applications to Cell/B.E. – ClustalW (*clustalw*) and FASTA (*ssearch34*). We detailed the efforts required, especially the changes required in each application for data movement to and from SPEs, which require explicit DMA commands, and provided analysis and results for our implementation. In this paper, we discuss our implementation of ClustalW (*clustalw*) using the SPE software cache support, which was introduced in Version 3.1 of the IBM SDK for MultiCore Acceleration [4] as an alternative to DMA commands for custom data movement. We describe the changes we made to utilize the compiler-supplied software data cache. In our experiences, the use of caches simplifies the program development and enhances the programmer productivity to a significant extent. We also compare the performance our cache implementation to the DMA version of the same application.

II. Cell/B.E.

The Cell/B.E. is a heterogeneous, multi-core chip optimized for compute-intensive workloads and broadband, rich media applications. The Cell/B.E. is composed of one 64-bit Power Processor Element (PPE), 8 specialized accelerators called Synergistic Processing Elements (SPEs), a high-speed memory controller and high-bandwidth bus interface, all integrated on-chip. The PPE and SPEs communicate through an internal high-speed Element Interconnect Bus (EIB). The memory interface controller (MIC) provides a peak bandwidth of 25.6GB/s to main memory. The Cell/B.E. has a clock speed of 3.2GHz and theoretical peak performance of 204.8 GFLOPS (single precision) and 21 GFLOPS (double precision).

The PPE is the main processor of the Cell/B.E. and is responsible for running the operating system and coordinating the SPEs. Each SPE consists of a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). The SPU is a RISC processor with 128 128-bit single-instruction-multiple-data (SIMD) registers and a 256KB Local Store (LS). The SIMD pipeline can run at four different granularities: 16-way 8b integers, 8-way 16b integers, 4-way 32b integers or single-precision floating-point numbers, or 2 64b double-precision floating point numbers.

The 256 KB local store is used to hold both the instructions and data of an SPE program. The SPU cannot access main memory directly. The SPU issues DMA commands to the MFC to bring data into the LS or write the results of a computation back to main memory. Thus, the contents of the LS are explicitly managed by software. The SPE can continue program execution while the MFC independently performs these DMA transactions.

III. Cache implementation on Cell/B.E.

At the hardware level, SPU can only directly access data that resides in the SPE's local store – it cannot directly access data in system memory. The SPU must use DMA operations to transfer data from main memory to local store in order to access it, and must issue DMA operations to transfer the results of computation back to main memory. However, a recent feature in the GCC and XLC compilers makes it possible for SPE programs to directly access data in main memory. This feature is called **Named Address Space support** or **ea address space support** [4]. It allows the programmer to add the type qualifier *__ea* to variable declarations in a SPE program to indicate that this variable actually resides in system memory. This allows the PPE and SPE portions of an application to share data as easily as two different functions within the PPE portion of the code.

The compilers are able to provide this functionality by generating the DMA operations for the referenced data under the covers. However, issuing a DMA for every access to an *__ea* variable could severely impact performance, so the compiler implements a software-managed cache in the local store of the SPE. The compiler allocates a region of local store and uses it to hold copies of *__ea* qualified variables accessed by the SPE program. Several implementations of software data caching for the SPEs have been developed, but we chose to explore the cache support in the compilers because it is nearly transparent to the application programmer.

In our work, we used the GCC compiler and implementation of named address space support [5]. The size of the cache allocated by *spu-gcc* can be set to 8KB, 16KB, 32KB, 64KB, or 128KB with the compiler option *-mcache-size=*, where the default size is 64KB. The compiler does not attempt to maintain coherence of its cached copies with the contents of system memory, so some care must be used in choosing data to access using this mechanism. However, the cache manager does track modifications to cached data at a byte level, and writes these changes back to system memory with an atomic read-modify-write operation, to prevent the loss of concurrent modifications to other bytes on the same cache line. If the programmer carefully partitions the data in system memory such that no two SPEs write to the same cache line, performance may be improved by avoiding the atomic DMAs in the update phase by specifying the *-mno-atomic-updates* compiler option.

If *__ea* variables are used for communication between threads, GCC provides the *__cache_evict(__ea void *ea)* function to flush updates from the cache back to system memory for a given *__ea* variable. Other SPEs that wish to see these updates must also evict the line from their cache before reading the new value. A form of synchronization, such as mailboxes or signotify registers, is typically needed to ensure the first thread completes its updates to memory before the second thread attempts to read the new value. GCC also provides a *__cache_touch(__ea void *ea)* function that can be used to prefetch data into the cache. These and a few other special cache functions are available in the *spu_cache.h* header file.

IV. ClustalW Cache Implementation

ClustalW is a progressive multiple sequence alignment application. The major time-consuming step of the ClustalW alignment is the all-to-all pairwise comparisons which can take 60%-80% of the execution time. ClustalW compares all input sequences against each other, thus performing a total of $\frac{n(n-1)}{2}$ alignments for n sequences. In [1], we explained in detail our Cell/B.E. implementation of ClustalW that uses explicit DMA operations to transfer data to and from the SPEs. This implementation executes the forward_pass function, which is the most time consuming portion of the all-to-all pairwise alignment, on the SPEs, dividing the sequences among the SPEs in a round-robin fashion. We developed our SPE version of forward_pass from an existing version that used Altivec functions by replacing the Altivec APIs with SPE vector APIs. We adopt the same strategy for our cache implementation, focusing on running the forward_pass function on the SPEs, and we have reused the bulk of our previously developed SPE-optimized forward_pass function. For more details on the kernel implementation of *forward_pass* and the parallelization of the step among the SPEs, please refer to [1]. In this section, we focus on the difference between the DMA and the cache implementation in data movement to and from SPEs.

In the DMA implementation, we had to make changes to the data layout in the PPU code, so that the data can be DMAed to and from the SPEs; DMA operations require the SPE address and the PPE address, as well as the size of data being transferred, must be a multiple of 16 bytes. In the original ClustalW implementation, the sequences were stored as an array of pointers, each pointer storing the address of a sequence; the sequences are all of varying lengths. For the DMA implementation, all of the sequences are packed into a single one-dimensional array; each sequence begins at a 16-byte aligned address, and is padded in length to the maximum sequence length rounded to a multiple of 16 bytes. The output of the forward pass function executed on two sequences must be placed into an array of structures in main memory. We modified this array so that each structure is aligned on a 16-byte boundary to allow the SPE to DMA the results directly into the appropriate structure. These changes allow every SPE to DMA in any 2 sequences and DMA out the output result obtained by pairwise alignment of the 2 sequences. The SPE recieves pointers to this combined sequence array, the output values array, and the sequence length array, which is needed for computation. The SPE also recieves the scalar values necessary for computation of penalties for gap-opening and gap-extension.

The cache implementation does not require any of these changes, as the *__ea* qualifier works for arbitrarily aligned variables stored in the PPE address space. Variables in the PPE code that are required for computation by the SPEs are declared with the *__ea* qualifier in the SPE code: this includes the original array of pointers, sequence length array, scalar values, and the alignment matrix. The only reason for quad-word alignment of the sequences was for DMAs to complete successfully, and was not a requirement for Altivec-enabled code. In certain cases, it was necessary to copy values of local function variables into global variables to make them properly accessible to the SPE through the __ea mechanism. With these changes, we could get the code to use SPEs for *forward_pass* computation.

We also experimented an alternate approach for moving data between main memory and the SPE local store. This approach uses the software caching mechanisms to create local copies of data in the SPE local store. The data to be copied is declared as an __ea variable, and the SPU code then simply copies data from this variable into a buffer in local store. This approach eliminates the need for 16-byte alignment restrictions which the DMA commands require, thus significantly enhancing programmer productivity.

Besides software caching mechanisms to copy data, we also experimented with data access and layouts on the PPE side and cache touch APIs to prefetch the sequences being pairwise aligned into the SPE local store. However, we did not find notable performance improvement from either of these techniques.

In Section V, we detail the performance of the cache implementation with varying input sizes and SPEs, and compare it with the DMA-only implementation from our previous work. We also pinpoint the reasons for the overhead of the cache implementation over the DMA only implementation.

V. Results

A. Experimental Setup

Our experimental setup consists of a QS20 blade with 2 Cell/B.E processors running at 3.2 Ghz and a total memory of 1 GB. Sequence similarity applications do not depend on double-precision floating-point performance to a large extent, so using the older QS20 is not a major issue with these applications. We used *ppu-gcc* and *spu-gcc* release 4.1.1 to compile the PPU and the SPE code respectively. The optimization level was fixed at *-O3* for both the PPU and the SPE code.

B. Results and Analysis

Figure 1 shows the time taken by ClustalW on 1 SPE, with the DMA code completed before, and the cache implementation for the pairwise alignment step. The figure shows the performance of the cache implementation with varying cache sizes of 8 KB and 64 KB(default). The input datasets *class-A*, *class-B* and *class-C* for ClustalW are included as part of the BioPerf package suite [6], downloaded from `www.bioperf.org`. As can be seen, the performance of the cache implementation is about 2X slower than the DMA implementation, for both cache sizes. The difference in performance of the cache implementation between the two cache sizes we tested is negligible, which indicates that the cache performance of ClustalW is not limited by the cache sizes, even for the largest inputs. As explained in Section IV, we have retained the original SPE kernel for pairwise alignment from the DMA implementation, thus we can conclude that the **difference in performance is entirely due to overhead of software cache over the DMA**.

To understand which data accesses are leading to the overhead in the cache implementation, we changed our DMA-only implementation to a mix of DMA and software cache. Figure 2 shows the difference in performance as we move from a DMA-only implementation to a mix of DMA and caches; the data was collected using 1 SPE only. The *DMA + Cache (Sequences)* data shows an implementation in which all data is DMAed into SPE, except for the DNA sequences, which are accessed through the software caches. The *DMA + Cache (Alignment Matrix)* data shows the implementation in which all data is DMAed into the SPEs, except for the alignment matrix; the alignment matrix is used to compute the scores of aligning every 2 characters of the query sequence and the library sequence. The *DMA + Cache (Alignment Matrix + Sequences)* data shows the implementation in which all data is DMAed into the SPEs except for the DNA sequences and the alignment matrix, which are accessed through the software cache. Lastly, the *Cache only* data shows the cache-only implementation with no DMAs. As can be seen, the difference in performance of the *Cache only* implementation and the *DMA + Cache (Alignment Matrix + Sequences)* implementation is negligible, so we can conclude that the **main overhead in the cache-only implementation is primarily due to the access of the alignment matrix and the DNA sequences**. Cached access to other data such as the scalar values for gap-extension and penalty, as well as storing the results matrix does not carry any major penalty.

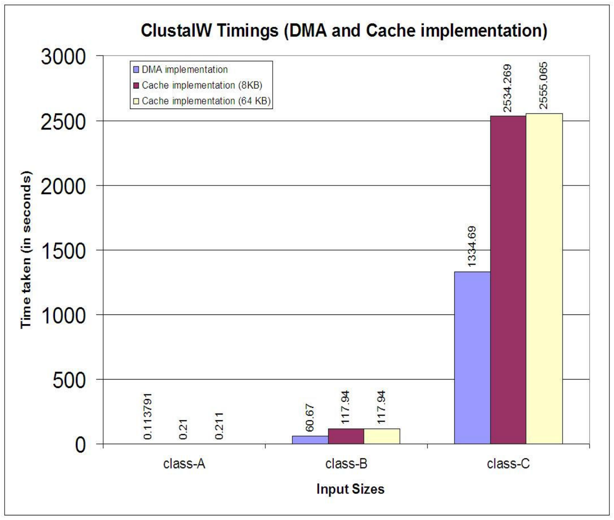

Fig. 1. Timings of ClustalW pairwise alignment step on a single SPE with DMA and initial cache version

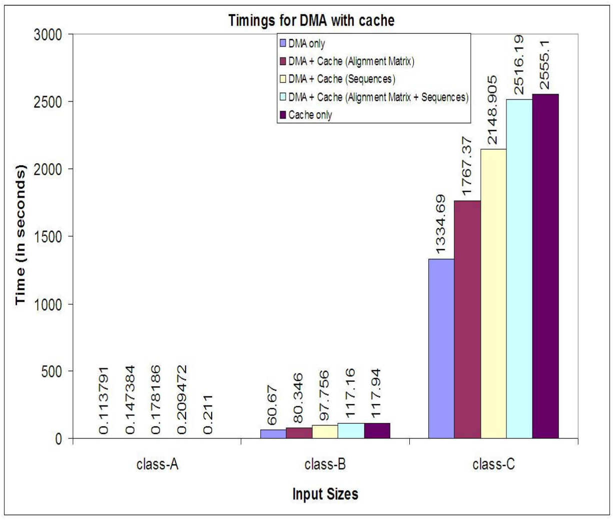

Fig. 2. Performance of DMA implementation with caches for sequences and alignment matrix

Since the main overhead of the cache implementation is in accessing the alignment matrix and the sequences through the cache, we experimented with copying the alignment matrix and the sequences into a buffer in local store, and then using this local copy in the SPE computation. The copy is performed using software caching mechanisms just before the SPE starts its computation. The size of the alignment matrix is 4096 bytes and independant of input size, so we perform the copy of the alignment matrix for all input sizes. However, we only perform the copy for the sequence data when it will fit within the buffer space allocated in the SPE local store. This is the case for the class-A and class-B inputs, whose sequences consume 2990 bytes and 71K bytes, respectively. But the sequences for the class-C input are 325K bytes, so for this input we fall back to accessing the sequences through the software cache.

Figure 3 shows the performance of the cache implementation when the alignment matrix and/or the sequences are copied to a local buffer prior to the SPE computation. This figure shows that the performance of the implementation that copies both the alignment matrix and sequences is nearly equivalent to the DMA implementation for the class-A and class-B inputs. Since the sequences in the class-C are to copy into a local buffer, we still see an overhead of more than 50% compared to the DMA implementation.

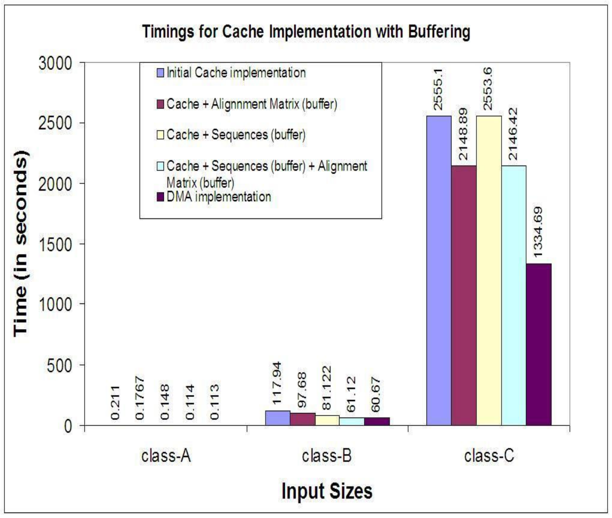

Fig. 3. Performance of cache implementation with copying of alignment matrix and sequences into a buffer before computation

Figure 4 shows the difference in performance using multiple SPEs upto a maximum of 16, for both the cache and the DMA implementations for class-B and class-C input of BioPerf. We implemented the same static load-balancing strategy as explained in [1] for both the implementations. As is evident from Figure 4, we can see that both implementations scale well with increasing number of SPEs. **The cache implementation does not lead to increased overhead at higher number of SPEs**, and shows the same overhead as the runs done with 1 SPE. These results are for cache sizes of 8 KB. For size-B input, in which we can copy the entire input and the alignment matrix into a local store buffer, the performance of the cache implementation is very close to the DMA implementation. For class-C input, in which only the alignment matrix can be copied entirely into the local store, we still see a performance overhead compared to the DMA implementtaion.

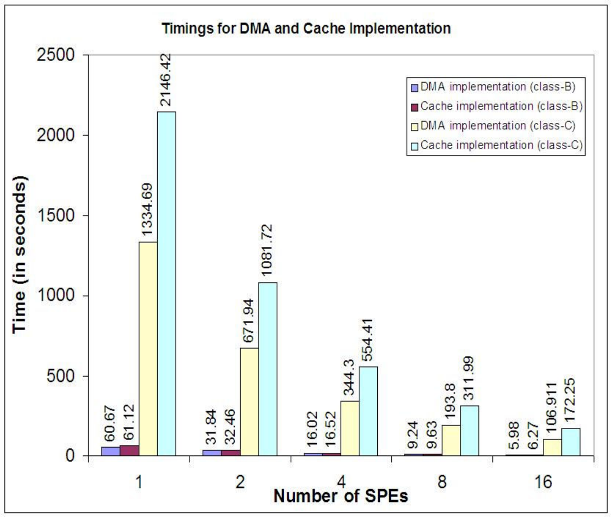

Fig. 4. Timings of ClustalW pairwise alignment step on multiple SPEs with cache and DMA versions

VI. Conclusions and Future Work

In this paper, we described our implementation of ClustalW running on Cell/B.E. that uses software caches inside the SPEs for data movement between PPE and SPEs. Our initial effort resulted in an overhead of about 2X compared to the DMA only version, but copying the data to the local store prior to using it results in a version which is very close to the DMA performance. Using the software caches enhances the programmer productivity to a significant extent, without a major decrease in a performance. For the largest input size, which does not fit into the SPE local store, we still see an overhead of more than 50% in performance compared to the DMA-only implementation. Our future work will involve investigating the sources of overhead of SPE caches and devising ways to reduce them. Such an implementation will also make it possible to use SPEs for pairwise alignment of very large DNA sequences, without a significant overhead.

References

[1] V. Sachdeva, M. Kistler, E. Speight, and T.-H. K. Tzeng, "Exploring the Viability of the Cell Broadband Engine for Bioinformatics Applications," in *Proc. of the Sixth IEEE Intl. Workshop on High Performance Computational Biology*, 2007.

[2] Y. Lui, W. Huang, J. Johnson, and S. Vaidya, "GPU Accelerated Smith-Waterman," in *Proc. GPGPU Workshop (GPGPU06)*, University of Reading, UK, May 2006, http://www.mathematik.uni-dortmund.de/~goeddeke/iccs/index.html.

[3] T. Oliver, L. Y. Yeow, and B. Schmidt, "High Performance Database Searching with HMMer on FPGAs," in *Proc. 6th Workshop on High Performance Computational Biology (HiCOMB 2007)*, Long Beach, CA, Apr. 2007.

[4] IBM, "IBM SDK for Multicore Acceleration," 2008, https://www.ibm.com/developerworks/power/cell/.

[5] M. R. Meissner, "Adding named address space support to the gcc compiler," in *Proc. 2009 GCC Developers Summit*, Montreal, Canada, jun 2009.

[6] D. Bader and V. Sachdeva, "An Open Benchmark Suite for Evaluating Computer Architecture on Bioinformatics and Life Science Applications," in *Proc.SPEC Benchmark Workshop 2006*, Austin, TX, Jan. 2006.