

CellMT: A Cooperative Multithreading Library for the Cell/B.E.

Vicenç Beltran, David Carrera, Jordi Torres and Eduard Ayguadé
Technical University of Catalonia (UPC) - Barcelona Supercomputing Center (BSC)
Barcelona, Spain
{vbeltran, dcarrera, torres, eduard}@ac.upc.edu

Abstract

The Cell/B.E. processor has proved that heterogeneous multi-core systems can provide a huge computational power with high efficiency for a wide range of applications. The simple design of the computational units and the use of small managed local memories is the key to achieve high efficiency and performance at the same time. However, this simple and efficient hardware design comes at the price of higher code complexity. The code written to run in this kind of processors must deal with several issues such as code vectorization, loop unrolling or the explicit management of local memories. Some of these issues such as vectorization or loop unrolling can be partially solved by the compiler, but the overlapping of data transfer and computation times must be manually addressed by the programmer with techniques such as double buffering that increase the code complexity. In this paper we present a user level threading library called CellMT that effectively hide memory latencies. The concurrent execution of several threads inside each SPU naturally overlaps computation and data transfer times without increasing the code complexity. To prove the suitability and feasibility of our multi-threaded library, we perform an exhaustive performance evaluation with a synthetic benchmark and a real application. The experimental results show that the multithreaded approach can outperform a hand-coded double buffering scheme, with speedups from 0.96x to 3.2x, while maintaining the complexity of a naive buffering scheme.

1. Introduction

The Cell/B.E. processor provides high computational power and memory bandwidth with a simple and efficient hardware design that overcomes the memory wall problem [1] with the use of software-managed local memories. The use of managed local memories instead of traditional caches is the most distinctive characteristic of these processors. Each local storage is directly accessible only from its own processor removing the need to implement coherency protocols across the local storage of each processor. This simplifies the hardware design and improves the scalability of the system. The performance improvement that can be obtained with this processor come at the cost of higher

software development complexity. To obtain the best performance of the Cell processor, the programmer should address all the issues which are common to other state of the art multi-core processors, such as code parallelization, code vectorization, loop unrolling, branch predication and data alignment. Current compiler technology can (partially) address most of the mentioned issues, but the Cell processors have an additional one to overcome, the programmer need to manually manage the data transfers between main memory and each local storage. The naive approach to this problem can be trivially implemented on current compilers but the performance obtained will be unacceptable because we also need to overlap the data transfer and computation times. The most well-known and widely used technique to overlap computation and data transfer times is the use of double or multi buffering schemes. These techniques are effective for regular applications with a predictable memory access pattern, but cannot be always applied. Moreover, double buffering or multi buffering techniques increase the code complexity and must be manually implemented by the programmer in a case by case basis, reducing the system productivity.

In this paper, we present and evaluate our CellMT cooperative multithreading library that naturally overlaps the computation time of one thread with the transfer time of other threads inside the same SPU. This library provides a cooperative multi-threading model. So it relies on the threads themselves to relinquish control once they are at a context switch point. This cooperative multi-threading model is a perfect fit for any processor with a managed local store, such as the Cell processor, because the context switch points are easily identified. In fact, all the applications written for the Cell have this points explicitly identified by the memory flow control (MFC) operations used to wait for DMA request or Mailbox messages. The CellMT library provides a familiar and well understood programming model that is similar to the model used to split work across SPUs, so it does not increase the complexity of the application. Moreover, this technique is more prevalent than double buffering techniques because it does not need to know the next DMA request to be performed in advance, hence it is specially well suited for applications with non predictable memory accesses.

The rest of the paper is organized as follows: Section 2 compares the available techniques to overlap computation

and data transfer times for the Cell/B.E. Section 3 introduces the Cell/B.E. architecture. Section 4 presents the CellMT threading library. Section 5 describes the benchmarks used to evaluate the performance characteristics of our threading library. Finally, Section 6 draws the conclusions and future work.

2. Related work

Techniques such as double-buffering or multi-buffering [2][3] have been widely used in the Cell/B.E. to manually hide DMA latencies and to overlap computation and data transfer times. Although both techniques are very effective, they must be used on a case-by-case basis, because these techniques require non-trivial and error-prone code modifications which are only suitable for applications with very predictable memory accesses. Other techniques have been proposed in the literature to hide memory latencies such as in [4], where the authors propose a prefetching technique for I/O intensive applications, which is only effective for applications with huge working sets that do not fit in main memory. In [5], a software cache is proposed that improves the performance of some specific applications with a irregular memory access pattern. In [6], the authors describe a programming framework that automatically manages the application data and uses an optimal buffering scheme that overlap the application computation and data transfer times. Like most of the other related work, this framework is only useful for applications with a predictable access pattern, furthermore we need to write the application from scratch in a new programming model. Although all the aforementioned techniques are valid and effective for some specific applications, we need a more general solution that can be effectively implemented without increasing the overall application complexity. To this end, we have investigated the use of multi-threading to hide memory latencies on the Cell/B.E. The SPUNK [7] nano-kernel provides a micro-threading model to increase the utilization of the Cell/B.E. resources. The main goal of SPUNK is also to overlap DMA latencies with computation, but its high context switch overhead of 4 ticks (compared to the 2.9 and 3.9 ticks of a DMA request of size 128 and 2048 bytes respectively) make it unsuitable for most applications. Additionally, SPUNK requires to rewrite the application to follow an event-driven model. In contrast, the CellMT library has a context switch overhead of only 0.7 ticks, which increases the scope of its applicability. The CellMT threading library provides a low level interface with a high degree of flexibility. It also furnishes a high level interface (a wrapper to the standard `libspe2`), that ease development of new applications and porting existing ones. This high level library only provides the illusion to the PPU code of more available SPEs, so that existing Cell/B.E. applications can easily be ported. With this high level library, we can run a Cell/B.E. application

with virtually no modifications on the PPU and SPE code. This makes it very attractive for any existing application that wants to make the most of the Cell/B.E. without increasing the code complexity. In addition to ease the development of end user applications, the CellMT library can also be used to simplify the implementation of runtime system and specialized programming frameworks such as [8], [9], [6] and [10].

3. Cell/B.E. architecture

The Cell Broadband Engine Architecture (CBEA) [11] is a single chip heterogeneous multiprocessor. The design goals of the Cell processor were to address the fundamental challenges facing modern microprocessor development: high memory latencies and on-core power dissipation. Until now, microprocessors have achieved performance improvements through higher clock frequencies and deeper pipelines, but the fundamental problem that current processors face is the memory wall [1]. On modern processors significant amounts of time are spent waiting in memory stall, due to the large difference between the processor and the memory speed. Large memory latencies make it difficult to obtain further performance gains with traditional processor designs based on hardware caches. The Cell processor approaches this problem in a different way, providing a heterogeneous processor with explicit memory management. This approach potentially improves the throughput of the processor, but also increase the effort to develop an application.

Figure 1 shows the three basic components of the Cell processor. First, the PowerPC Processor Element (PPE), which is primarily intended to manage global resources. Second, the Synergistic Processing Elements (SPE) that are specialized vector processors with a private local storage and a DMA engine, which can perform up to 16 asynchronous DMA transfers between the local storage and main memory at the same time. Finally, the communication between the PPE, the SPEs, main memory, and external devices is realized through an Element Interconnect Bus (EIB). The EIB has a theoretical peak data bandwidth of 204.8GB/s, but the DMA operations with main memory are limited to 25.6GB/s. Moreover, the data transfer times from main memory to a SPU local storage have a latency of at least 1000 processor cycles that is not negligible for small data transfers. The following results have been measured with the *dmabench* utility provided by the IBM SDK for the Cell/B.E.

Figure 2 shows in a log-log scale the transfer time of DMA read operations with block sizes that ranges from 8 bytes to 16 KBytes in the x-axis. The y-axis measures the transfer time in processor cycles. The transfer time of a DMA read operations are composed of a initial delay plus the DMA block size divided by the memory bandwidth. This initial delay dominate the transfer times of DMA read operations for block sizes of up to 1024 bytes. In Figure 2

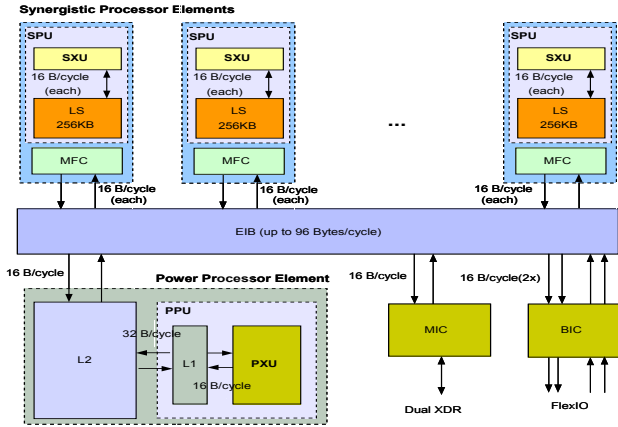


Figure 1. The cell broadband engine architecture.

there are three different configurations evaluated. The first measures the performance of DMA operations when only one SPU is active, the second configuration measures the performance of DMA read operations of eight concurrent SPUs. Finally, the last configuration shows the performance when all the 16 SPUs of a QS20 are evaluated. As we can see, the bus congestion increases the latency from 1000 cycles for one SPU alone to more than 3000 for the 16 SPUs configuration. The initial DMA transfer delay is not amortized until we use DMA block sizes of at least 2048 bytes, when the total time is dominated by the data transfer time.

The data presented on Figure 2 shows the need to overlap computation time and transfer time, specially for transfer sizes of less than 1024 bytes. For instance, in the 16 SPEs configuration, when a SPU issue a DMA get operation of less than 1024 bytes, the processor will be waiting at least 3000 cycles until the data is ready on the local storage, which is unacceptable for most applications. This data also shows the opportunity to improve this situation with the use of multi-threading inside each SPE. As we will describe in detail in the next section the key idea behind our threading library is to perform a fast thread context switch to make the most of these wasted SPU cycles. Our vision is that with the use of a cooperative multi-threading model on a processor with a managed local storage we can leverage the benefits of this simple and high performance hardware design, but with a programming complexity similar to a chip multithreading design (CMT) such as the Niagara processor [12].

4. CellMT library

The cooperative multi-threading library is implemented on a core library that provides all the features and flexibility required to run complex multi-threaded application inside the SPUs. This core library, which is described in detail in

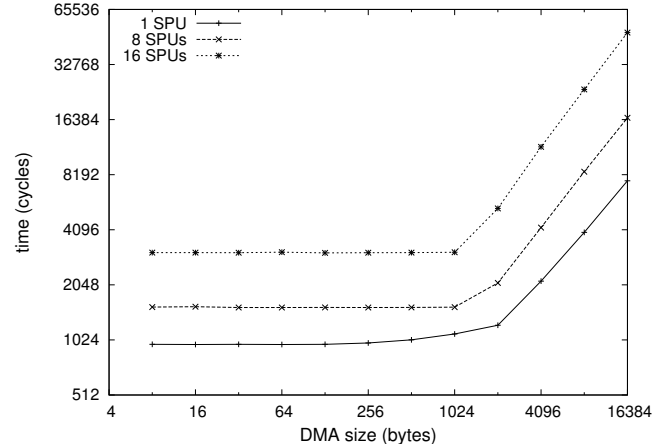


Figure 2. DMA latencies of the Cell/B.E.

section 4.1, provides a low-level threading API that can be directly called from the SPU application code. This low-level API is useful to write applications with complex interactions between threads, but its flexibility can also increase the complexity of the applications itself. To address this issue, the CellMT library also provides an additional library described in section 4.2, that simplifies the development of applications that follow a common threading pattern. This additional library is a wrapper to the standard `libspe2` library that is used from the PPU side, and provides a high level abstraction to use the SPU threads. For the sake of clarity, we have omitted the implementation details of both libraries, but the source code can be downloaded from [13]. Finally, section 4.3 presents two implementations of the same encryption kernel to compare the complexity of a double buffer scheme and our multithreaded approach. In [14] there is a detailed description of the library APIs, as well as, some additional code examples.

4.1. The libcellmt library

This library contains the three basic functions required to run a cooperative multi-threaded application. The most important functions are `run_thread(...)`, `wait_for(...)` and `yield()`. The first one allows the programmer to spawn a new thread that will start executing a specified function with a list of parameters and using its own stack space. This function returns an error if the number of threads has reached the maximum or 0 on success. The thread id of the new thread is returned on the `int *th_id` parameter. The function `wait_for(...)` is used to wait for the end of a previously created thread. Finally, the `yield()` function is used to transfer the execution to another thread. The main task carried by this function is to save the value of the PC register and the other 48 non-volatile registers of the current thread, and restore the same registers with the values of the next thread to execute. We use a

round-robin algorithm to schedule the next thread to run. This simple algorithm minimizes the overhead of the `yield()` function, which is around 170–180 cycles for any number of active threads. Besides these three functions, there are other auxiliary functions such as `get_thread_id()` that returns the thread id of the current thread or `get_free_stack_space()` that returns the available stack space of the current thread. Finally, there are several non-blocking functions which can be used to wait for common events such as DMA completions or SPU channel activity. Currently the maximum number of SPU threads supported are 16, although this number can be easily increased if necessary. The library is completely embedded with the user application at compilation time and it does not need to initialize any dynamic data structure or variable.

4.2. The `libspe2mt` library

This library is intended to ease the development and porting of applications that are already designed to run across multiple SPUs, which is the case of most Cell/B.E. applications. It follows the philosophy of the standard `libspe2` library, but extends its functionality to support the execution of multiple threads in each SPE. From the point of view of the PPU code, it is like if there were more available SPUs to run on. This library provides the same function as the original `libspe2` library, as well as an additional one: the `spe_mt_context_add_thread(...)` which can be used to specify the number of threads that will run on a `spe_mt_context`. Each of the configured thread will execute the `main(...)` function of this `spe_mt_context` with its own specified arguments in a transparent way. If we use this library, we only need to do minor modifications to the original PPU code, while in the SPU code just have to change the macro or function used to wait for DMA operations (see the next section for an illustrative example).

4.3. Multithreading vs. double buffering

In this section we compare the code complexity of our multi-threaded approach vs. the code complexity of a double buffering scheme. Listing 1 shows the simplest code required to encrypt a data buffer resident in main memory. As we can see, the steps required to encrypt a buffer of an arbitrary size are straightforward. In general, the original buffer will not fit in the private memories available in the SPUs, so we need to split it into smaller blocks. The original buffer is transferred to the local storage in blocks of size `lbuffer`. Each of these blocks are then encrypted on the local storage and the resulting data is copied back to main memory. This process is repeated until all the data has been encrypted. The main drawback of this code is that we are not overlapping data transfer and computation times. Notice that this example follows one of the simplest and most common processign

patterns used on the Cell/B.E. processor that we will call "get-compute-put" from now on.

Listing 1. AES simple buffering

```
void aes_simple_buffering(
    unsigned long long buffer,
    const unsigned int bsize,
    unsigned int lbuffer,
    const unsigned int lbuffer_size,
    const AES_KEY *key,
    const int mode){

    const int iters = bsize/lbuffer;
    int tag = mfc_tag_reserve();

    for(int i=0; i<iters; i++){
        mfc_getb(lbuffer, buffer, lbuffer_size, tag, 0, 0);
        mfc_wait_all(tag);
        AES_ecb_encrypt_fast(lbuffer, lbuffer,
            lbuffer_size, key, mode);
        mfc_put(lbuffer, buffer, lbuffer_size, tag, 0, 0);
        buffer += lbuffer;
    }

    mfc_wait_all(tag);
    mfc_tag_release(tag);
}
```

To improve the performance of the code shown in Listing 1 we can use a double buffering scheme. With double buffering we can overlap the computation time of the current block with the transfer time of the next block. Listing 3 shows the double buffering version of the original code. As we can observe, the complexity of the loop has increased. Now we need an epilogue and a prologue to correctly process the first and last blocks of the buffer. Moreover, the loop must be unrolled to process two blocks per iteration. We also need two times more space in the local storage to allocate the buffers required to do double buffering. Although this code is more efficient than the first version, it is also more complex and error-prone.

Listing 2. `mfc_wait_all` vs. `mfc_wait_mt_all`

```
inline void mfc_wait_all(const int tag){
    mfc_write_tag_mask(1 << tag);
    mfc_read_tag_status_all();
}

inline void mfc_mt_wait_all(const int tag){
    const unsigned int mask = 1 << tag;
    unsigned int ret;

    do {
        yield();
        mfc_write_tag_mask(mask);
        mfc_write_tag_update(
            MFC_TAG_UPDATE_IMMEDIATE);
        ret = mfc_read_tag_status();
    } while (unlikely((ret & mask) == 0));
}
```

Listing 3. AES double buffering

```

void aes_double_buffering(
    unsigned long long buffer,
    const unsigned int bsize,
    unsigned int *lbuffer[2],
    const unsigned int lbsize,
    const AES_KEY *key,
    const int mode){

    const int iters = (bsize/lbsize)/2;
    int tag[2] = {mfc_tag_reserve(),
                 mfc_tag_reserve()};
    mfc_get(lbuffer[0], buffer+(lbsize*0),
            lbsize, tag[0],0,0);
    mfc_get(lbuffer[1], buffer+(lbsize*1),
            lbsize, tag[1],0,0);
    for(int i=0; i<iters-1; i++){
        mfc_wait_all(tag[0]);
        AES_ecb_encrypt_fast(lbuffer[0],
                              lbuffer[0], lbsize,
                              key, mode);
        mfc_put(lbuffer[0],buffer+(lbsize*0),
                lbsize, tag[0], 0, 0);
        mfc_getb(lbuffer[0],buffer+(lbsize*2),
                 lbsize, tag[0], 0, 0);
        mfc_wait_all(tag[1]);
        AES_ecb_encrypt_fast(lbuffer[1],
                              lbuffer[1],
                              lbsize, key, mode);
        mfc_put(lbuffer[1],buffer+(lbsize*1),
                lbsize, tag[1], 0, 0);
        mfc_getb(lbuffer[1],buffer+(lbsize*3),
                 lbsize, tag[1], 0, 0);
        buffer += (2*lbsize);
    }
    mfc_wait_all(tag[0]);
    AES_ecb_encrypt_fast(lbuffer[0],
                          lbuffer[0],
                          lbsize, key, mode);
    mfc_put(lbuffer[0],buffer+(lbsize*0),
            lbsize, tag[0], 0, 0);
    mfc_wait_all(tag[1]);
    AES_ecb_encrypt_fast(lbuffer[1],
                          lbuffer[1],
                          lbsize, key, mode);
    mfc_put(lbuffer[1],buffer+(lbsize*1),
            lbsize, tag[1], 0, 0);
    mfc_wait_all(tag[0]);
    mfc_tag_release(tag[0]);
    mfc_wait_all(tag[1]);
    mfc_tag_release(tag[1]);
}

```

Finally, the multi-threaded implementation is like the code presented in Listing 1, but the call to the *mfc_wait_all(tag)* function is replaced by a call to the *mfc_mt_wait_all(tag)* function. The rest of the code remains completely unchanged. Listing 2 shows the differences between these functions. Both functions are always called immediately after a DMA operation has been started. The original function issues a blocking instruction that waits for the completion of a specific DMA operation, so the whole processor becomes

stalled. On the other hand, the multi-threaded version calls the *yield()* function to instantly block the current thread and change the execution to another thread. At some point, another thread will again call the *yield()* function and the original thread will resume its execution. Then, the thread will issue a non-blocking instruction to check if the DMA has been completed. If this is the case the thread will continue its execution, otherwise the thread will call the *yield()* function again.

5. Evaluation

We have used two different benchmarks to evaluate the performance of our CellMT library: the first is a synthetic benchmark that can be parametrized to generate different workloads; the second benchmark is the AES encryption kernel available in the IBM SDK 3.1, which is used to verify the correctness of the results obtained with the synthetic benchmark.

All the experiments have been conducted on a QS20 blade powered with two Cell processors clocked at 3.2 GHz with 1 GB of RAM. The default Linux kernel (version 2.6.22-5) and a virtual page size of 4 KBytes is used in all the experiments. The experiments have been executed several times to obtain results with a low standard deviation.

5.1. Synthetic benchmark

The synthetic benchmark was created to capture the performance characteristics of the most representative processing pattern used in the Cell/B.E. processor, which is the "get-compute-put" pattern (already presented in the example of Section 4.3). In this general processing pattern, a portion of the input data is transferred to the local storage, then the data is locally processed and the resulting output data is written back to the main memory. The performance characteristics of an application that follows this pattern is mainly determined by two factors: the data transfer size and the operational intensity of the processing algorithm. The operational intensity of an algorithm is usually defined as the number of *flops* per byte of input data [15]. This definition is useful to compare the performance of a given kernel across a number of different hardware architectures. In the scope of this paper, we define the operational intensity of an algorithm as the number of cycles spent for each byte of input data, because we are evaluating the performance of a set of kernels on the same hardware architecture.

We have developed a synthetic benchmark that follows the above-mentioned "get-compute-put" processing pattern, but with parametric data transfer sizes and operational intensities. The synthetic benchmark is designed to process an input buffer resident on the main memory. The PPU side of the program calculates the boundaries of the buffer splits to be processed by each SPU. Each SPU receives the

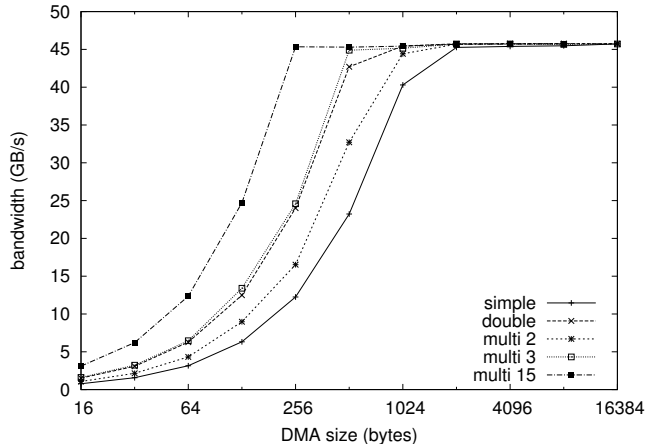


Figure 3. Read only performance.

length and the size of the buffer to be processed. Each SPU splits its input buffer in blocks of $DMA\ size$ and applies a processing algorithm with an operational intensity $cycles_per_byte$. Once one block of data is processed it is copied back to main memory, and the process is repeated until all the input data has been processed.

There are three different implementations of the synthetic benchmark to make the actual processing: simple buffering, double buffering and the multi-threading approach. In order to produce the most accurate results, the parametrization of the kernel is done at compile time with the help of some macros. In this way we avoid the introduction of any run-time overhead into the processing algorithm.

5.2. Synthetic benchmark: read-only

The first experiment is aimed to measure the maximum memory bandwidth that can be achieved by each of the three versions (simple buffering, double buffering and multithreading) of our synthetic benchmark. In this case the operational intensity is 0 cycles/byte, and the data is not copied back to main memory, so we are only measuring the raw read performance of each version. That is, the upper bound of the synthetic benchmark in terms of actual data transfer capacity. The size of the input buffer is 512MB, so each SPU will process a split of 32MB. Note that the peak bandwidth of the QS20 blade used in the experiments (powered by 2 Cell/B.E. processors) is 51.2GB/s, as it was discussed in Section 3.

Figure 3 shows the performance of the synthetic benchmark when parametrized with 0 $cycles_per_byte$, and $DMA\ size$ values ranging from 16B to 16KB. Notice that in this experiment data is not processed nor sent back to main memory. Note that the x-axis is in a log scale. The y-axis shows the effective memory bandwidth achieved by the synthetic benchmark. Results show the performance

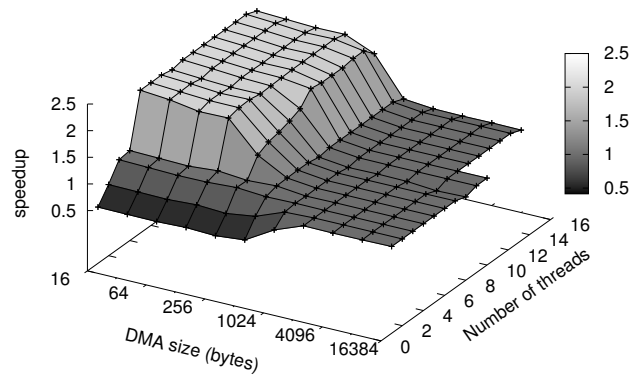


Figure 4. Speedup multithreading vs double buffering.

of simple buffering, double buffering and multi-threading approach; the multi-threaded version is evaluated with 2, 3 and 15 threads. As it can be seen, the 3 threads multi-threaded configuration gets the same performance as the double buffering scheme, while with 15 threads it clearly outperforms the other configurations. The maximum effective performance observed in the experiments for all configurations is around 45 GB/s, what is very close to the theoretical peak performance of a dual Cell/B.E. blade, that is 51.2 GB/s. The difference between the achieved bandwidth and the theoretical peak performance can be explained by the SPU TLB thrashing that is observed for data-sets larger than 16MB. Running the same experiments with an input buffer size of 16MB, we observed a sustained bandwidth of 50GB/s.

As it can be observed, for $DMA\ size$ above 2048 bytes, all configurations deliver the same performance because the communication becomes bandwidth intensive (see Figure 2). For configurations with less than 2048 bytes of $DMA\ size$, the actual bandwidth is determined by the number of DMA requests on-the-fly. Notice that the multi-threaded scheme configured with 3 threads obtains the same performance as the double-buffering configuration (and clearly above the single buffering configuration), while with 2 threads the performance is between that delivered by the single-buffering and double-buffering configurations. This result can be explained with the fact that multi-threaded has slightly higher overhead when compared to the double-buffering scheme. The extra overhead is due to the light thread context switches. Notice that with 15 threads the multi-threaded configuration is able to obtain an effective bandwidth higher than the other configurations for values of $DMA\ size$ below 1024 bytes.

In a second experiment, still using a 0 $cycles_per_byte$ configuration, we compare all the configurations of the multi-threaded approach (from 1 to 16 threads) with the

double-buffering scheme. Notice that the maximum number of threads to be used is limited by the number of on-the-fly DMA requests supported by one SPU (see Section 4 for more details). Results for this experiment can be seen in Figure 4 in terms of speedup. As it can be observed, only configurations using 1 or 2 threads deliver lower performance than the double buffering scheme, but only by a small margin. For all the configurations with a DMA transfer size larger than 512 bytes the performance are the same, but for smaller DMA transfer speedups of up to 2.4x can be observed. Although it can hardly be seen in the Figure 4, performance for configurations with 2, 4, 8 and 16 threads are slightly lower than the performance obtained for similar configurations using a different number of threads. This can be warranted based on the unbalanced DMA requests to the different memory banks in these configurations.

Finally, note that some data points in Figure 4 are missing. This is caused by the fact that for the 16KB *DMA size* configurations and more than 10 threads, the amount of memory needed to store the data, code and DMA buffers exceeds the 256KB capacity of the local storage.

5.3. Synthetic benchmark: read-compute-write

In this section we compare the multithreaded, double buffering and simple buffering approaches. We have evaluated the synthetic benchmark with a comprehensive combination of configurations, comprising different values of *DMA size* and *cycles_per_byte*. These experiments follow the "get-compute-put" processing pattern already explained.

From the results presented in Section 5.2, we observed that using 15 threads in the multithreaded technique produces the higher performance for small values of *DMA size*, and the same performance for the rest of configurations. Therefore, this is the configuration of threads we used in the experiments presented in this Section.

In Figures 5, 6 and 7 we can see the performance of the simple buffering, double buffering and multithreading approach respectively. All figures uses a log scale in both x and y axis, but notice that the axis showing the *cycles_per_byte* starts at 0, which is used to represent in a convenient way the difference between a computational kernels with 1 *cycles_per_byte* and no data processing at all. The three configurations have a similar performance shape, with low performance for configurations with small *DMA size* or high *cycles_per_byte* and higher performance for configurations with higher *DMA size* and lower *cycles_per_byte*. The three configurations reach the peak performance around 20GB/s of data processed, that is equivalent to 40GB/s of memory bandwidth between the main memory and the local storage (as now we write back to the memory all the read data from the input buffer). Although the performance shape of the three configurations is similar, the multithreaded approach has a higher number of configurations that reach

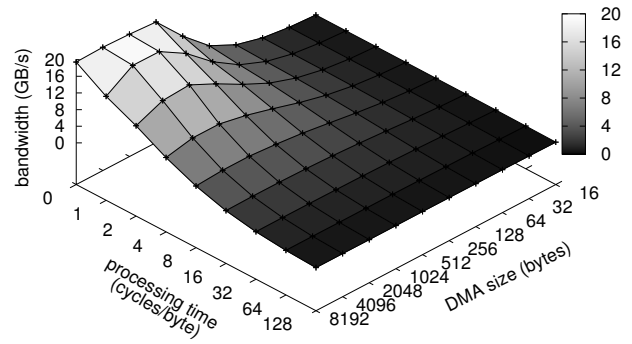


Figure 5. Simple buffering performance.

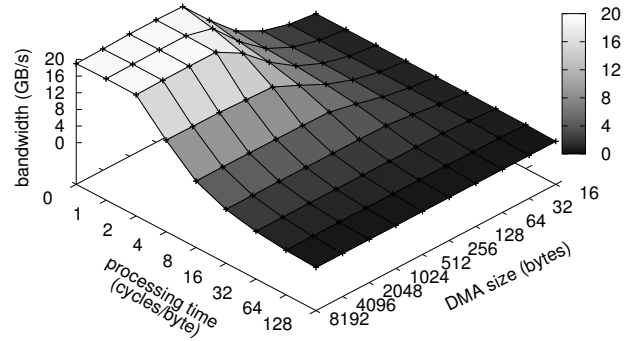


Figure 6. Double buffering performance.

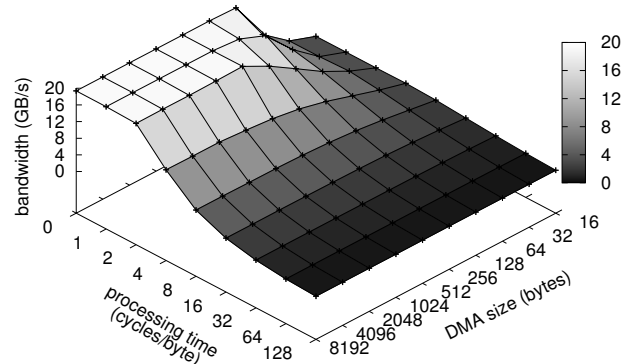


Figure 7. Multithreading performance. (15 threads)

the peak performance, followed by the double buffering configuration.

To better understand the performance difference of the three configurations we have plotted the speedup between the double buffering and the simple buffering scheme, and between the multithreading approach and the double buffering scheme. Figures 8 clearly shows the better performance of the double buffering scheme. As we can see we obtain a 2x speedup for a large set of configurations and the same performance for the configurations that combine a large *DMA size* with a high *cycles_per_byte*.

Figure 9 shows how multithreading techniques improve the performance of double buffering by a factor higher than 3x for the configurations with a *DMA size* of 256 bytes or less and a computational intensity (*cycles_per_byte*) of 4 cycles/byte and less. For the rest of configurations, the speedup decreases as the *DMA size* or the *cycles_per_byte* increase, until it converges with the performance of the double buffering scheme (0.96x speedup).

5.4. AES encryption kernel

In this section we use a real computational kernel to validate the results obtained with the synthetic benchmark. Therefore, we expect to meet the performance of at least the double buffering scheme keeping the programming complexity of a simple buffering scheme. For this purpose we have used the AES encryption kernel provided by the IBM SDK 3.1, which follows the same "get-compute-put" processing pattern used in the synthetic benchmark. Moreover, this kernel is well suited to work with data block of arbitrary size, so it can be easily used with different values of *DMA size*.

Figure 10 shows the performance of the AES encryption algorithm for the three evaluated configurations: simple buffering, double buffering and multi-threading (with 15 threads). Notice that the x-axis is in a log scale. The computational ratio of this AES implementation –with a 128 bits encryption key– is of 13 cycles/byte, so it should behave between the 8 and 16 cycles/byte configuration of the synthetic benchmark. As we can expect, the simple buffer configuration has the worst performance, while the double buffering and the multi-threaded configuration are very close. The multithreaded configuration is better for small values of *DMA size*, while the double buffer configuration is slightly better for values of *DMA size* between 128 and 512 bytes. The measured operational intensity of this encryption function is 13 cycles/byte for *DMA size* larger or equal than 128 bytes, but for small values of *DMA size* the function call overhead –this function cannot be inlined– is not negligible, so the operational intensity of the encryption kernel increases. Therefore, it is not feasible to directly compare the synthetic benchmark and the AES kernel for small values of *DMA size*.

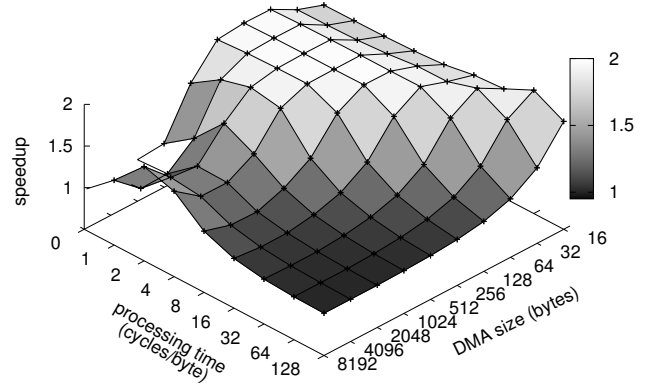


Figure 8. Speedup double buffering vs simple buffering.

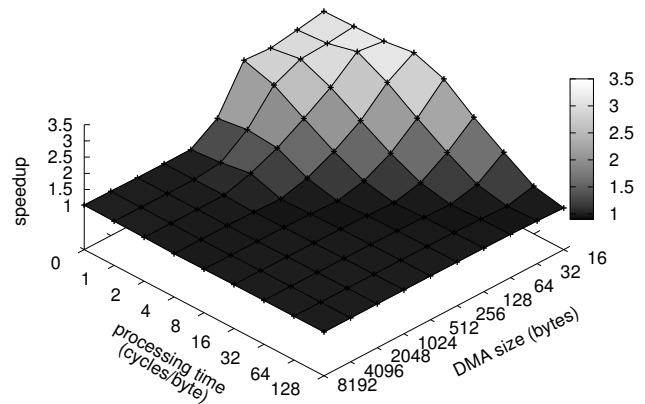


Figure 9. Speedup multithreading vs double buffering. (15 threads)

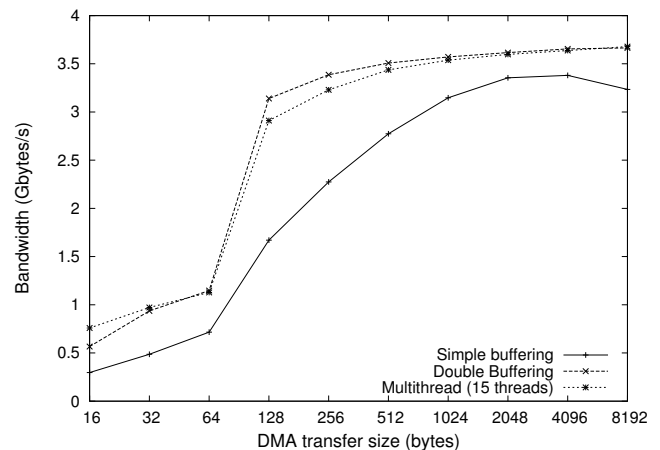


Figure 10. AES encryption performance. (16 SPUs)

6. Conclusions and future work

In this paper we have presented and evaluated our CellMT threading library with promising results. This library provides a cooperative user-level threading model that can be used to run multi-threaded applications inside the SPUs of the Cell/B.E. processor. These multi-threaded applications naturally overlap the computation time of one active thread with the data transfer times of other blocked threads, without requiring the use of limited and error-prone techniques such as double buffering schemes that can only be applied on applications with a very predictable memory access pattern. We have implemented a synthetic benchmark to evaluate the suitability and performance of the CellMT library in a insightful and exhaustive way. The speedup of the multi-threaded implementation range from 0.98x to 6.6x compared to the baseline implementation, and from 0.96x to 3.2x compared to the double buffering implementation. In summary, with the help of the CellMT library we can write programs with the complexity of a naive buffering scheme and better performance than using a double buffering scheme. In the future we plan to further investigate the suitability and feasibility of our threading library for applications with non predictable memory access such as list ranking and other combinatorial algorithms that cannot even use double buffering techniques. We will also try to further reduce the current context switch overhead to improve the performance of our library for even a wider range of applications, as well as, to integrate the CellMT threading library with other runtime systems supporting novel programming models, such as CellSs [8] and MapReduce [9] [16], or future implementations of OpenCL [10].

Acknowledgements

Many thanks to Xavier Martorell and Marc Gonzalez for valuable discussions during the course of this work. This work is partially supported by the Ministry of Science and Technology of Spain (contract TIN2007-60625), the Generalitat of Catalunya (2009-SGR-980), the European Commission in the context of the HiPEAC2 network of excellence (contract no. ICT-217068) and the SARC project (contract no. 27648), and the MareIncognito project under the BSC-IBM collaboration agreement.

References

- [1] W. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," University of Virginia, Charlottesville, VA, USA, Tech. Rep., 1994.
- [2] T. Chen, Z. Sura, K. O'Brien, and J. K. O'Brien, "Optimizing the Use of Static Buffers for DMA on a CELL Chip," in *LCPC*. Springer Berlin / Heidelberg, 2006, pp. 314–329.
- [3] Jonathan Bartlett, "Programming high-performance applications on the Cell/B.E. processor," April 2007, <http://www.ibm.com/developerworks/library/pa-linuxps3-6/>.
- [4] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos, "Dma-based prefetching for i/o-intensive workloads on the cell architecture," in *CF '08: Proceedings of the 2008 conference on Computing frontiers*. New York, NY, USA: ACM, 2008, pp. 23–32.
- [5] T. Chen, T. Zhang, Z. Sura, and M. Gonzalez, "Prefetching irregular references for software cache on cell," in *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2008, pp. 155–164.
- [6] B. Bouzas, R. Cooper, J. Greene, M. Pepe, and M. J. Prelle, "MultiCore Framework: An API for Programming Heterogeneous Multicore Processors," in *Proceedings of First Workshop on Software Tools for Multi-Core Systems*. New York, NY, USA: Mercury Computer Systems, 2006.
- [7] M. F. Ahmed, R. A. Ammar, and S. Rajasekaran, "Spenc: adding another level of parallelism on the cell broadband engine," in *IFMT '08: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*. New York, NY, USA: ACM, 2008, pp. 1–10.
- [8] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a Programming Model for the Cell BE Architecture," *SC Conference*, vol. 0, p. 5, 2006.
- [9] Y. Becerra, V. Beltran, D. Carrera, M. González, J. Torres, and E. Ayguadé, "Speeding up distributed mapreduce applications using hardware accelerators," in *38th International Conference on Parallel Processing (ICPP)*, 2009.
- [10] Khronos OpenCL Working Group, "Open Computing Language (OpenCL)," December 2008, <http://www.khronos.org/opencv/>.
- [11] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its first implementation," *IBM DeveloperWorks*, November 2005.
- [12] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.
- [13] Barcelona Supercomputing Center (BSC), "Project CellMT Homepage," July 2009, <http://sourceforge.net/projects/cellmt/>.
- [14] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé, "Using Cooperative Multithreading on the Cell BE," Computer Architecture Department, Technical University of Catalonia, Tech. Rep., April 2009. [Online]. Available: http://gsi.ac.upc.edu/reports/2009/27/tr_cellmt.pdf
- [15] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
- [16] M. de Kruijf and K. Sankaralingam, "MapReduce for the Cell B.E. Architecture," Department of Computer Sciences, The University of Wisconsin-Madison, Madison, WI, Tech. Rep. TR1625, 2007.