

MultiCore Framework: An API for Programming Heterogeneous Multicore Processors

Brian Bouzas, Robert Cooper, Jon Greene,
Michael Pepe, Myra Jean Prella.
Mercury Computer Systems, Inc.
199 Riverneck Road
Chelmsford, MA 01824

Abstract

This paper describes a software toolkit for programming heterogeneous multicore processors, such as the IBM Cell Broadband Engine (BE)TM processor, that contain explicit non-cached memory hierarchies. Emerging heterogeneous multicores contain specialized processing elements that do not possess traditional cache hardware. Instead, multiple levels of the memory hierarchy must be explicitly managed by software. The MultiCore Framework (MCF) toolkit provides an abstract view of this hardware oriented toward computation of multidimensional data sets. High performance and ease of programming are the primary goals. Through the toolkit, the programmer has explicit control over how data and processing is divided up among processing and memory elements, while being insulated from specific hardware details. Further, MCF separates data organization from computation. This enables the problem size and resource allocation to be changed without touching highly optimized inner loop code.

Multicore Processors

To continue the dramatic improvement in computing power achieved over the last few decades, processor architects have embraced the idea of combining multiple processing elements on a single chip. Modest approaches comprise two to four general-purpose processor (GPP) cores in symmetric multiprocessing mode – essentially an SMP on a chip. In order to achieve the highest performance for a given chip area and power budget, more-aggressive multicore architectures provide larger numbers of simpler processing elements [1, 2]. These simpler cores reduce hardware complexity, consume less chip area, draw lower power, and often provide higher raw

performance per core. However, the simplified architecture and higher performance come at the price of a dramatically increased burden on software.

Heterogeneous multicores combine one or more GPP cores with more-numerous specialized cores. This configuration enables conventional software to run on the GPP while the heavy computational portions of a problem can be mapped onto the specialized cores.

The Cell BE [3] couples a general-purpose PowerTM architecture Processing Element (PPE) with eight specialized cores, termed Synergistic Processing Elements (SPEs), which are oriented to vector and matrix computation. The SPE contains a 128-bit wide vector unit, 128 registers each of 128 bits, and 256K bytes of memory, termed the *local store*. The SPE's load and store instructions can reference only local store. To access memory in the rest of the system, each SPE possesses a DMA unit called the Memory Flow Controller (MFC). The SPE instruction set includes instructions that queue DMA commands to the MFC; these DMA commands are executed in parallel with normal computation on the SPE.

The local store, though minimal in size, is considerably larger (4 to 8 times) than the L1 data cache on most general-purpose processors. The difference is that because there is no cache management hardware to move code and data to and from main memory, software must explicitly manage the memory hierarchy.

Parallel N-Dimensional Matrix Computation on Multicore Architectures

Computation of vector and n -dimensional matrix problems is often highly regular and is readily amenable to parallelization. The key task is to decide how to process the data in parallel in a way that minimizes memory bandwidth and hides memory latency. Typically, a processing element is assigned a subdivision of the entire data set and proceeds to *strip mine* that data, that is, process blocks of input data to produce blocks of output data. At other points in the algorithm, output blocks containing intermediate results must be reorganized so that processing elements can strip mine the data

in an efficient way during the next phase of processing. Matrix transpose is an example.

For the programmer, mapping an algorithm in this way to the details of a heterogeneous multicore architecture is a challenging task. To be successful, the vast majority of programmers must be insulated from the complexities of the chip architecture.

We believe that, in the near term, the most practical way to achieve this is through libraries and the *function off-load* model. In this model, the primary application logic executes on the GPP and calls functions that execute on specialized processor cores for parts of the problem demanding high performance. A smaller group of programmers will program these specialized functions. These programmers' demands are relatively uncompromising from the performance point of view. They are looking for any kind of improvement in programmer productivity, but not at the expense of performance. It is for them that Mercury developed the MultiCore Framework.

MultiCore Framework

The MultiCore Framework (MCF) is an API for programming a heterogeneous multicore chip oriented toward n -dimensional matrix computation. An MCF application consists of a manager program (that runs on the Cell's PPE) and one or more workers (SPEs on the Cell).

MCF's most important features include the ability to:

- define teams of worker processes,
- assign tasks to a team of workers,
- load and unload functions (called plugins) by worker programs,
- synchronize between the manager and worker teams,
- define the organization and distribution of n -dimensional data sets through data distribution objects, and
- move blocks of data via channels.

This last feature is perhaps the most important. It facilitates multi-buffered strip mining of data between a large memory (XDR memory on the Cell) and small worker memories (SPE local store on the Cell) in such a way as to insulate worker code from the details of the data organization in main memory.

Distribution Objects and Channels

The key abstractions in MCF are the distribution object and the channel. The distribution object is created by the manager, and specifies all the attributes of how data is dimensioned and tiled. It defines attributes of a manager *frame* (stored in main memory) and a worker *tile* (stored in worker memory). A frame will contain an entire data set for one execution of an algorithm. A tile is the

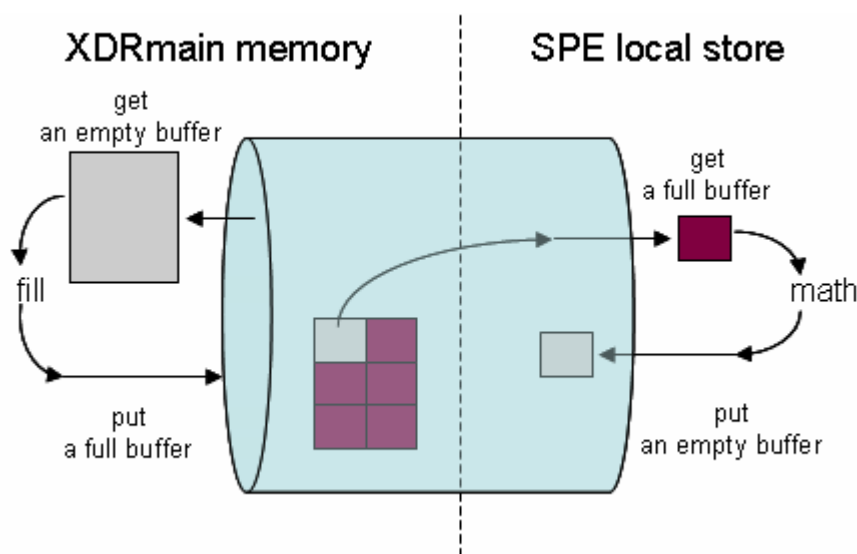


Figure 1. An input tile channel

subdivision of a frame that is processed at one time by a given worker.

MCF *tile channels* provide multi-buffered strip mining between a large main memory and the small worker memories. A distribution object must be supplied when a tile channel is created in order to define the dimension, size and other parameters of the channel's frames and tiles.

Once the data organization has been defined by the manager, the worker task simply connects to the tile channel and thereby obtains the prescription for DMA transfers that move data into or out of the local store. If desired, application code in the worker can inspect the tile descriptor to learn the tile dimension, size and other tile-specific parameters.

Figure 1 represents a frame in XDR memory being divided into six tiles, each of which will be consumed one at a time by a single SPE via an input tile channel. The manager program obtains an empty frame buffer from the channel, fills it, and puts it back into the channel.

The worker program takes one tile buffer at a time from the input channel. While it owns the buffer,

the worker program is free to perform whatever math operations are required by the algorithm. Once it has no further use for the data in the buffer, it returns it "empty" to the tile channel. At this point, the tile channel implementation is able to issue a DMA command to prefetch the next tile into SPE local store.

The output channel operates in an analogous manner, with the worker program obtaining an empty tile buffer, filling it, and returning it to the channel.

When the worker program attaches to a channel, one of the arguments specifies the number of local tile buffers to use, and this in turn controls how much prefetching or double buffering of tiles takes place.

To support data reorganization (such as global matrix transpose) MCF also contains *reorg channels*. Although they are not discussed further in this paper, reorg channels are similar to tile channels. On the Cell processor, they support coordinated many-to-many transfers among SPEs.

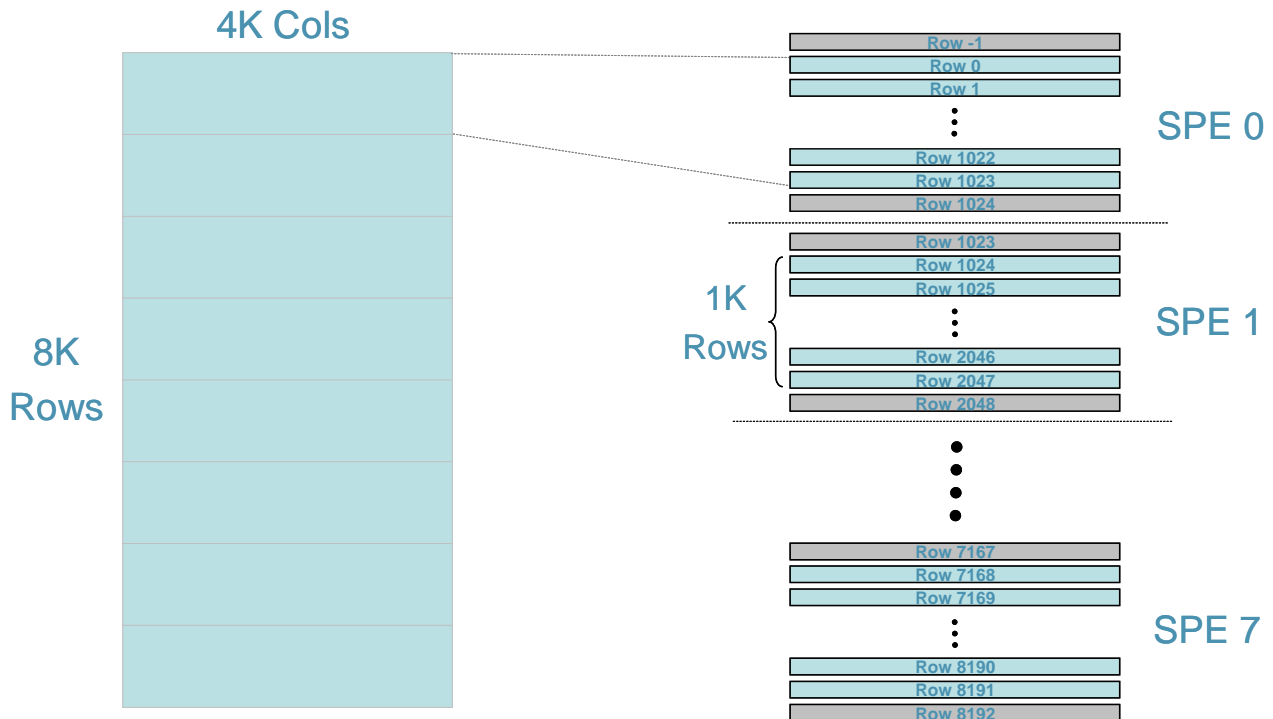


Figure 2: Image filter frame and tile organization

```

#define INPUT_CHANNEL 0 // "name" for input and
#define OUTPUT_CHANNEL 1 // output channels

main(int argc, char **argv)
{
    mcf_m_net_create(n_workers); // specify number of workers (SPEs)
    mcf_m_net_initialize(); // launch MCF kernel on workers

    mcf_m_net_add_task("worker_filter.spe", &task); // load SPE from file system executable to XDR
    mcf_m_team_run_task(MCF_ALL_WORKERS, task); // run task on each worker

    mcf_m_tile_distribution_create_2d( // specify source distribution
        n_frame_elems_x, n_frame_elems_y, // XDR frame size
        n_tile_elems_x, n_tile_elems_y, // tile size
        &distrib_src);
    mcf_m_tile_distribution_set_assignment_overlap( tile_src, ...); // specify overlap between partitions
    mcf_m_tile_distribution_create_2d( ..., &distrib_dst); // specify destination distribution

    mcf_m_tile_channel_create( // channel "name"
        INPUT_CHANNEL, // will connect to all workers
        MCF_ALL_WORKERS, // use source distribution object just created
        distrib_src, // flag specifying direction of channel
        &tile_channel_src); // source channel
    mcf_m_tile_channel_create( // destination channel
        OUTPUT_CHANNEL, MCF_ALL_WORKERS,
        distrib_dst, MCF_TILE_CHANNEL_DEST,
        &tile_channel_dst);
    mcf_m_tile_channel_connect( tile_channel_src );
    mcf_m_tile_channel_connect( tile_channel_dst );

    mcf_m_tile_channel_get_buffer(tile_channel_src, &src_frame ); // get XDR address of buffer to fill
    // code to fill frame with input image goes here
    mcf_m_tile_channel_put_buffer(tile_channel_src, &src_frame ); // make data available to workers

    mcf_m_tile_channel_get_buffer(tile_channel_dst, &dst_frame_desc ); // wait for results
    mcf_m_team_wait(MCF_ALL_WORKERS); // wait for each team member's task to exit
}

```

Figure 3: Manager program for image filter

```

mcf_w_main (int n_bytes, void * p_arg_ls)
{
    mcf_w_tile_channel_create( INPUT_CHANNEL, &tile_channel_src);
    mcf_w_tile_channel_create( OUTPUT_CHANNEL, &tile_channel_dst );
    mcf_w_tile_channel_connect( tile_channel_src );
    mcf_w_tile_channel_connect( tile_channel_dst );

    mcf_w_tile_channel_get_buffer(tile_channel_src, &in[0]); // get first two rows
    mcf_w_tile_channel_get_buffer(tile_channel_src, &in[1]);
    while ( ! mcf_w_tile_channel_is_end_of_channel ( tile_channel_src ))
    {
        mcf_w_tile_channel_get_buffer(tile_channel_src, &in[2]); // get third row
        mcf_w_tile_channel_get_buffer(tile_channel_dst, &out); // get an "empty" output buffer
        // code to compute 3x3 filter on input rows goes here
        mcf_w_tile_channel_put_buffer(tile_channel_dst, out); // start moving results back to XDR
        mcf_w_tile_channel_put_buffer(tile_channel_src, in[0]); // put "empty" buffer back into channel
        in[0]=in[1];
        in[1]=in[2];
    }
}

```

Figure 4: Worker program for image filter

To better illustrate MCF's capabilities, we use the example of a 3x3 convolution filter applied to a 32 megapixel image consisting of 4K columns by 8K rows. In essence, the filter takes a 3x3 patch of input pixels to produce a single output pixel. The work is divided among the eight SPEs such that each SPE produces 1024 contiguous output rows (Figure 2). Since three rows of input are required to produce one output row, each SPE takes 1026 input rows. (Padding is used to extend the image with one additional row at the beginning and end.)

At any given time, each SPE has three complete rows of 4K pixels to which it applies the filter 4094 times to produce one output row. In order to overlap transfers to and from main memory with the filter computation, four input buffers are used in a circular fashion along with two output buffers for double buffering. Each buffer contains a single input or output row.

Abbreviated versions of the manager and worker programs are shown in Figures 3 and 4. The manager program, after initializing the MCF network, loads the object code for the worker task into main memory (`mcf_m_net_add_task()`) and then runs one copy of that task in each of eight SPEs (`mcf_m_team_run_task()`).

Next, the manager makes several calls to define the organization of the input and output data. The `mcf_tile_distribution_create_2d()` function creates a distribution object which describes how the data in main memory is organized, and how it is split up into tiles which can be worked on by SPEs. In this case, the arguments specify the 4K by 8K frame and the 4K by one row tile size. The following call to `mcf_m_tile_distribution_set_assignment_overlap()` would specify that there shall be one row of overlap between the tile allocations to each SPE. That means that SPE 1, for instance, will receive rows 1023 through 2048 (with zero-based indexing). This call would also specify that two rows (one at index -1, the other at index 8192) should be created (in this case by copying rows 0 and 8191 respectively) to handle the boundary cases.

The distribution object offers a rich set of attributes:

- number of dimensions in a frame of data,
- size of each dimension of a frame,
- size of each dimension of a tile,

- packing order of each dimension (i.e. the array indexing order); any permutation is supported,
- organization of compound data types (complex float, RGB triples). Data is interleaved when the components of a given compound data item are adjacent. Data is split if like-components of different data items are organized in contiguous blocks (e.g. for complex data, all the real components stored contiguously followed by all the imaginary components).
- partitioning policy across workers. This provides a means for the manager program to define how tiles are assigned to workers, for example, bands of columns, bands of rows, bands of planes, round robin, or all tiles to every worker. Additional properties are provided, such as partition overlap used in this example.

Returning to the example, the next four function calls create input and output tile channels and connect to the manager's endpoint of both.

Next, the manager obtains an empty frame from the input tile channel, initializes it with an 8K by 4K pixel frame and then puts the full frame back into the channel.

Finally the manager program waits for all members to complete. The rest of the program would presumably use the resulting image in the output frame. Code to clean up allocated objects in an orderly fashion is not shown.

The worker program does not contain any tile or channel definition calls. Instead, it simply creates the source and destination channels and connects to them. These calls result in the necessary information on tile sizes and the associated DMA commands being loaded into SPE local store by the MCF library.

Next, the worker enters the processing loop. To initialize the loop, it obtains the first two data buffers (input rows) from the source channel. Inside the loop it obtains the third source buffer (input row) and an empty output buffer (output row). The filter function is then called to compute the single output row from the three input rows. At the bottom of the loop, the empty source buffer is returned to

the source channel and the full destination buffer is pushed into the destination channel. Finally the input buffer descriptors are rotated in preparation for the next iteration of the loop which will bring in the next input row.

The loop exits when after the last tile within the source frame has been provided.

Experience

Mercury has implemented MCF on its Dual Cell-Based Blade and used it in the implementation of a 64K parallel FFT [4] and in porting existing customer applications to the Cell processor. In our implementation, the MCF worker kernel occupies 12KB of the 256KB local store.

Related Work

MCF builds on our experience with distributed memory multiprocessor architectures that support remote DMA. In essence, many of today's heterogeneous multicore architectures can be viewed as distributed memory multiprocessors on a chip. MCF is thus based on our previous work with the Parallel Acceleration System (PAS) [5] and the Data Reorg standard [6].

There are many potential approaches to managing explicit memory hierarchies automatically in software, including sophisticated compiler optimization [7], software implemented caching and stream oriented programming languages[8]. We chose a relatively practical approach which provides abstractions for data tiling common to highly optimized parallel n -dimensional matrix computation.

An area for future investigation is whether automatic compilation techniques and specialized languages can benefit from targeting libraries such as MCF.

Conclusion

MCF supports programmers that need fine-grained control of data distribution and assignment of processing resources on a multicore processor, but relieves them of the hardware-specific details involved.

The keys to achieving high performance, when specifically considering the Cell processor, are to:

- minimize SPE code – ideally just DMAs and math,
- overlap computation with DMA to/from the local store via tile channels, and
- exploit SPE-to-SPE bandwidth via reorg channels (don't make all transfers go through XDR).

MCF achieves these goals while providing the ability to modify problem size, number of workers, and other aspects of data and processing assignment without requiring any changes to the worker task.

Acknowledgements

We thank Miriam Leeser, Leigh McLeod and the anonymous reviewers for providing valuable feedback on earlier drafts of this paper.

References

- [1] M. Taylor, *et al.* Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams, In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, Munich, Germany, June, 2004. ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/raw_isca_2004.pdf
- [2] USC Information Sciences Institute, Advanced Systems Division. *A MORphable Networked microARCHitecture. Program Overview*, August 2002. <http://www.isi.edu/asd/monarch/>
- [3] T. Chen, R. Raghavan, J. Dale, E. Iwata. *Cell Broadband Engine Architecture and its first implementation*. November 2005. <http://www.ibm.com/developerworks/power/library/pa-cellperf/>
- [4] J. Greene, R. Cooper. A Parallel 64K Complex FFT Algorithm for the IBM/Sony/Toshiba Cell Broadband Engine Processor. In *Technical Conference Proceedings of the Global Signal Processing Expo (GSPx)*, 2005.
- [5] J. Greene, C. Nowacki, M. Prella. PAS: A Parallel Applications System for Signal Processing Applications. *International Conference on Signal Processing Applications and Technology*, 1996.

[6]DRI-1.0 Specification, September 25, 2002

<http://data-re.org/>

[7] A. Eichenberger, *et al.* Optimizing Compiler for the CELL Processor. In *Proceedings of 14th International Conference on Parallel Architectures and Compilation Techniques*, August 2005.

<http://caq.csail.mit.edu/crq/papers/eichenberger05cell.pdf>

[8] R. Rabbah, *et al.* High-Productivity Stream Programming For High-Performance Systems. In *Proceedings of the 9th Annual High Performance Embedded Computing Workshop (HPEC)*, September 2005.

<http://web.mit.edu/rabbah/www/docs/rabbah-hpec-2005.pdf>