Programming Heterogeneous Multicore Systems Using Threading Building Blocks*

George Russell[1], Paul Keir[2], Alastair F. Donaldson[3], Uwe Dolinsky[1], Andrew Richards[1], and Colin Riley[1]

[1] Codeplay Software Ltd., Edinburgh, UK
firstname@codeplay.com
[2] School of Computing Science, University of Glasgow, UK
pkeir@dcs.gla.ac.uk
[3] Computing Laboratory, University of Oxford, UK
alad@comlab.ox.ac.uk

Abstract. Intel's Threading Building Blocks (TBB) provide a high-level abstraction for expressing parallelism in applications without writing explicitly multi-threaded code. However, TBB is only available for shared-memory, homogeneous multicore processors. Codeplay's Offload C++ provides a single-source, POSIX threads-like approach to programming *heterogeneous* multicore devices where cores are equipped with private, local memories—code to move data between memory spaces is generated *automatically*. In this paper, we show that the strengths of TBB and Offload C++ can be combined, by implementing part of the TBB headers in Offload C++. This allows applications parallelised using TBB to run, without source-level modifications, across all the cores of the Cell BE processor. We present experimental results applying our method to a set of TBB programs. To our knowledge, this work marks the first demonstration of programs parallelised using TBB executing on a heterogeneous multicore architecture.

1 Introduction

Concurrent programming of multicore systems is widely acknowledged to be challenging. Our analysis is that a significant proportion of the challenge is due to the following:

Thread management: It is difficult to explicitly manage thread start-up and clear-down, inter-thread synchronization, mutual exclusion, work distribution and load balancing over a suitable number of threads to achieve scalability and performance.

Heterogeneity: Modern multicore systems, such as the Cell [1], or multicore PCs equipped with graphics processing units (GPUs) consist of cores with differing instruction sets, and contain multiple, non-coherent memory spaces. These heterogeneous features can facilitate high-performance, but require writing duplicate code for different types of cores, and orchestration of data-movement between memory spaces.

Threading Building Blocks (TBB) [2] is a multi-platform C++ library for programming homogeneous, shared-memory multicore processors using parallel loop and reduction operations, pipelines, and tasks. These constructs allow the user to specify what

* This work was supported in part by the EU FP7 STREP project PEPPHER, and by EPSRC grant EP/G051100/1.

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 117–125, 2011.

```
void SerialUpdateVelocity() {
  for(int i=1; i<Height-1; ++i)
    for(int j=1; j<Width-1; ++j)
      V[i][j] = D[i][j]*(V[i][j]+L[i][j]*
        (S[i][j]-S[i][j-1]+T[i][j]-T[i-1][j]));
}
```

Fig. 1. A serial simulation loop

can be safely executed in parallel, with parallelisation coordinated behind-the-scenes in the library implementation, thus addressing the above *thread management* issues.

Offload C++ [3] extends C++ to address *heterogeneity*. Essentially, Offload C++ provides single source, thread based programming of heterogeneous architectures consisting of a host plus accelerators. Thread management must be handled explicitly, but code duplication and movement of data between memory spaces is handled automatically. Offload C++ for the Cell processor under Linux is freely available [4].

In this paper, we combine the strengths of TBB and Offload C++ by implementing the crucial TBB *parallel for* construct. This allows applications that use these constructs to run, *without source-level modifications*, across *all* cores of the Cell BE architecture.

We also discuss data-movement optimisations for Offload C++, and describe the design of a portable template-library for bulk data-transfers. We show that this template-library can be integrated with TBB applications, providing optimised performance when Offload C++ is used on Cell. We evaluate our approach experimentally using a range of benchmark applications. In summary, we make the following contributions:

- We describe how an important fragment of TBB implemented using Offload C++ allows a large class of programs to run across all the cores of the Cell architecture
- We show how performance of TBB programs on Cell can be boosted using a *portable* template-library to optimise data-movement
- We demonstrate the effectiveness of our techniques experimentally

2 Background

The TBB `parallel_for` construct. We illustrate the `parallel_for` construct using an example distributed with TBB that simulates seismic effects. Figure 1 shows a serial loop. In Figure 2 the loop body is expressed as a C++ function object whose **`operator`**`()` method can process elements in a given range. The `parallel_for` function template takes a function object and an iteration space parameter. When invoked, the function object is applied to each element in the iteration space, typically in parallel. The programmer specifies neither the number of threads nor tasks.

Offload C++. The central construct of Offload C++ is the *offload block*, a lexical scope prefixed with the **`__offload`** keyword. In the Cell BE implementation of Offload C++, code outside an offload block is executed by the host processor (PPE). When an offload block is reached, the host creates an accelerator (SPE) thread that executes the code inside the block. This thread runs asynchronously, in parallel with the host thread. Multiple SPE threads can be launched concurrently via multiple offload blocks. Each offload block returns a handle, which can be used to wait for completion of the associated SPE thread. For full details, see [3].

```
struct UpdateVelocityBody {
  void operator()( const blocked_range<int>& r ) {
    for(int i=r.begin(); i!=r.end(); ++i)
      for(int j=1; j<Width-1; ++j)
        V[i][j] = D[i][j]*(V[i][j]+L[i][j]*
          (S[i][j]-S[i][j-1]+T[i][j]-T[i-1][j]));
} };
void ParallelUpdateVelocity() {
  parallel_for(blocked_range<int>(1, Height-1), UpdateVelocityBody());
}
```

Fig. 2. Simulation loop body as a C++ function object, executable using `parallel_for`

3 Offloading TBB Parallel Loops on the Cell BE Architecture

The example of Figure 2 demonstrates the ease with which TBB can parallelise regularly structured loops. TBB does not however support heterogeneous architectures such as the Cell BE. We now show that, by implementing the `parallel_for` construct in Offload C++ we can allow the code of Figure 2 to execute across *all* cores of the Cell. The key observation is that TBB tasks are an abstraction over a thread-based model of concurrency; of the kind provided by Offload C++ for heterogeneous architectures.

We implement the parallel loop templates of TBB to distribute loop iterations across both the SPE and PPE cores of the Cell. These template classes are included in a small set of header files compatible with the Offload C++ compiler. Figure 3 shows a simple version of `parallel_for`, while `parallel_reduce` can be implemented similarly.

The implementation in Figure 3 performs static work division. Multiple distinct implementations with different static and dynamic work division strategies across subsets of the available cores can be achieved via additional overloads of the `run` function. Dynamic work division is achieved by partitioning the iteration space dynamically to form a work queue, guarded by a mutex, from which the worker threads obtain work units. This provides dynamic load balancing, as workers with less challenging work units are able to perform more units of work. Overloaded versions of `parallel_for` allow the user to select a specific work partitioner, *e.g.* to select static or dynamic work division.

Work division between the SPE cores *and* the PPE core is performed in the `run` method of the `internal::start_for` template. Offload's automatic call graph duplication makes this straightforward, despite the differences between these cores: in Figure 3, `local_function` is called on both the SPE (inside the offload block) and PPE (outside the offload block) without modification to the client code.

In Figure 3, `NUM_SPES` holds the number of SPEs available to user programs. To use all the cores, we divide work between `NUM_SPES+1` threads. One thread executes on the PPE, the others on distinct SPEs. The body of `run` spawns offload threads parameterised with a sub-range and the function object to apply; it then also applies the function object to a sub-range on the PPE, before finally awaiting the completion of each offload thread.

When passing function objects into class and function templates, the methods to invoke are known statically. Therefore, the Offload C++ compiler is able to automatically compile the function object **`operator`**`()` routine for both SPE and PPE, and generate the data transfer code needed to move data between global and SPE memory [3].

```
template<typename Range, typename Body>
void parallel_for( const Range& range, const Body& body ) {
  internal::start_for<Range,Body>::run(range,body);
}

template<typename Range, typename Body>
struct start_for<Range, Body> {
  static void run( const Range& range, const Body& body ) {
    typedef Range::const_iterator iter;

    unsigned NUM_SPES     = num_available_spes();
    iter start            = range.begin(); // Simple 1D range work division
    iter end              = range.end();
    iter chunksize        = (end - start)/(NUM_SPES+1);
    offloadThread_t handles[NUM_SPES];
    const Body local_body = body;

    for (int i = 0; i < NUM_SPES; ++i) {
      iter local_begin = start + chunksize*i;
      iter local_end   = local_begin + chunksize;

      if (local_end > end) local_end = end;
      Range local_range(local_begin,local_end);
      handles[i] = __offload(local_body, local_range) { // Sub-range offloaded
        local_body(local_range);                        // to SPE for
      };                                                // asynchronous execution
    }
    {   // PPE also executes a sub-range
        iter local_begin = start + chunksize*NUM_SPES;
        Range local_range(local_begin,end);
        local_body(local_range);
    }
    for (int i = 0; i < NUM_SPES; i++)
      offloadThreadJoin(handles[i]); // Await completion of SPE threads
} };
```

Fig. 3. An Offload C++ implementation of `parallel_for` for the PPE and SPE cores

4 Portable Tuning for Performance

Offload C++ enables code written for a homogeneous shared-memory multicore architecture to run on heterogeneous multicore architectures with fast local memories. A consequence of this is that the relative cost of data access operations differs, depending on the memory spaces involved. We discuss both the default data-movement strategy employed by Offload; and the portable, manual optimisations we develop to tackle this.

Default data-movement: software cache. The compiler ensures that access to data declared in host memory results in the generation of appropriate data-movement code. The primary mechanism for data-movement on Cell is DMA. However, issuing a DMA operation each time data is read or written tends to result in many small DMA operations. This can lead to inefficient code, since providing standard semantics for memory accesses requires synchronous DMA transfers, introducing latency into data access.

A software cache is used to avoid this worst-case scenario. When access to host memory is required, the compiler generates a cache access operation. At runtime, a synchronous DMA operation is only issued if the required data is not in the software cache. Otherwise, a fast local store access is issued. When contiguous data is accessed, or the same data is accessed repeatedly, the overhead associated with cache-lookups is ameliorated by eliminating the far greater overhead associated with DMA. Writes to

global memory can be buffered in the cache and delayed until the cache is flushed or the cache-entry is evicted to make room for subsequent accesses.

The software cache is small: 512 bytes by default. The cache is a convenience, and can significantly improve performance over naïve use of DMA. However, accessing the cache is significantly more expensive than performing a local memory access, even when a cache hit occurs. For bulk transfers, where each cache-line is evicted without being reused, the cache leads to overhead without benefit.

Local shadowing. A common feature of code offloaded for Cell without modification is repeated access to the same region of host memory by offloaded code. In this case, rather than relying on the software cache, a better strategy can be to declare a local variable or array, copy the host memory into this local data structure *once*, and replace accesses to the host memory with local accesses throughout the offloaded code. If the offloaded code modifies the memory then it is necessary to copy the local region back to host memory before offload execution completes. We call this manual optimisation *local shadowing*, as illustrated below with a fragment of the raytracer discussed in §5.1:

```
Sphere spheres[sphereCount]; // Allocated in host memory
__offload {
  RadiancePathTracing(&spheres[0], sphereCount, ... );
};
```

The scene data in the `spheres` array, allocated in host memory, is passed into the `RadiancePathTracing` function, which repeatedly accesses its elements using the software cache. We can instead apply local shadowing by copying scene data from `spheres` into a locally-allocated array, `local`, declared within the offload block:

```
Sphere spheres[sphereCount]; // Allocated in host memory
__offload {
  Sphere local[sphereCount]; // Allocated in local memory
  for (int i = 0; i < sphereCount; ++i)
    local[i] = spheres[i];
  RadiancePathTracing(&local[0], sphereCount, ... );
};
```

A pointer to `local` is now passed to `RadiancePathTracing`, redirecting accesses to scene data to fast, local memory. This optimisation reduces access to scene data via the software cache to the "copy-in" loop; after this, accesses are purely local.

Local shadowing does not compromise portability: in a system with uniform memory the copy-in and copy-out are unnecessary, but yield equivalent semantics. Assuming that the code using the locally shadowed data is substantial, the performance hit associated with local shadowing when offloading is not applied is likely to be negligible.

Bulk data transfers. Offload C++ provides a header-file library of portable, type-safe template classes and functions to wrap DMA intrinsics and provide support for various data access use cases. Templates are provided for read-only (`ReadArray`), write-only (`Write-Array`) and read/write (`ReadWriteArray`) access to arrays in host memory.

The array templates follow the Resource Acquisition is Initialisation (RAII) pattern [5], where construction and automatic destruction at end of scope can be exploited to perform processing. Transfers into local memory are performed on construction of `ReadArray`/`ReadWriteArray` instances, and transfers to host memory are performed on destruction of `ReadWriteArray`/`WriteArray` instances.

```
struct UpdateVelocityBody {
  void operator()( const blocked_range<int>& range ) const {
    for( int i=range.begin(); i!=range.end(); ++i ) {
      ReadArray      <float, Width> lD(&D[i][0]), lL(&L[i][0]), lpT(&T[i-1][0]),
                                    lS(&S[i][0]), lT(&T[i][0]);
      ReadWriteArray<float, Width> lV(&V[i][0]);
      for( int j=1; j < Width-1; ++j )
        lV[j] = lD[j]*(lV[j]+lL[j]*(lS[j]-lS[j-1]+lT[j]-lpT[j]));
} } };
```

Fig. 4. Using DMA template wrappers for efficient data transfer

Figure 4 illustrates optimising the example of Figure 2 with bulk transfers. The declaration `ReadArray<float, Width> lD(&D[i][0])` declares `lD` a local **`float`** array, of size `Width`, and issues a synchronous DMA to fill `ld` with data from host array `D` (hence `lD` stands for "local `D`"). The `ReadWriteArray` instance `lV` is similar, except that when destroyed (on scope exit), a synchronous DMA restores the contents of `lV` to `V`. Velocity update is now performed with respect to local arrays only.

Bulk transfer templates share similarities with local shadowing. However, they hide details of copy-in and copy-out operations, and bypass the software cache completely, which is often significantly more efficient than an element-by-element copy would be.

At compile time, when targetting the PPE, a zero-copy template instantiation is invoked instead. This implementation is also usable on systems with single memory spaces, maintaining portability of code using the templates. Additional data-movement use cases can be implemented by users using the same template functions abstracting transfer operations used to implement the array templates.

Automation. The Offload C++ compiler provides feedback on memory access patterns which can guide the manual application of local shadowing and bulk data transfers. In principle, the compiler could conservatively perform these optimisations automatically, given good general-purpose heuristics for when such transformations are beneficial.

5 Experimental Evaluation

We demonstrate the effectiveness of our approach using a set of parallel TBB programs. Experiments are performed on a Sony PlayStation 3 (with six SPEs accessible), running Fedora Core 10 Linux and IBM Cell SDK v3.0. Parallel benchmarks are compiled using Offload C++ v1.0.4. Serial versions of the benchmarks are compiled using both GCC v4.1.1, and Offload C++ v1.0.4. The faster of the two serial versions is taken as the baseline for measuring the speedup obtained via parallelisation. Optimisation level `-O3` is used in all cases.

- **Seismic simulation.** Simulation discussed in §2 for a 1120×640 pixel display
- **SmallPT-GPU Raytracer.** Global illumination renderer generating 256×256 pixel images from scenes of 3 to 783 spheres, computing sphere-ray intersections with specular, diffuse, and glass reflectance with soft shadows and anti-aliasing [6]
- **Image processing kernels.** A set of 8 kernels operating on a 512×512 pixel image, performing black-and-white median, colour median and colour mean filtering; embossing; sharpening; greyscale conversion; Sobel and Laplacian edge detection

- **PARSEC Black-Scholes.** Partial differential equations modelling the pricing of financial options, from the PARSEC benchmark suite [7] using the *large* data set
- **PARSEC Swaptions.** Simulates pricing a portfolio of swaptions using the Heath-Jarrow-Morton and Monte Carlo methods; from PARSEC using the *large* data set

5.1 Results

We present results showing the performance increases obtained by parallelising each benchmark across all available cores of the Cell (6 SPEs + PPE), compared with PPE-only execution. In some cases, the speedup using all cores is more than 7×. The SPE cores are significantly different to the PPE, so we would not expect them to be directly comparable; a specific program may run faster across the SPEs due to higher floating point performance, or efficient use of scratch-pad memory.

Seismic Simulation: After an initial offload of the original code, we found that the data transfer intensive nature of this code results in non-optimal performance on the SPE as the data being processed is still held in the global memory, and not in fast SPE local store. To address this, we used the `ReadArray` and `ReadWriteArray` templates, as shown in Figure 4, to obtain a 5.9× speedup over the PPE alone.

Image Processing Kernels: Figure 5 shows performance results. We used local shadowing to hold input pixel rows in stack allocated arrays, implementing a sliding window over the input image. Fetching a new pixel row would over-write the local buffer storing the oldest, and here we utilised our bulk data transfer template operations. Writes of individual output pixels were also buffered, and written out via bulk transfer.

SmallPT-GPU Raytracer: Figure 6 shows performance results for three versions of the SmallPT raytracer in raytracing six scenes compared to the serial baseline. The first version uses `parallel_for` to execute on the SPEs and PPE. The second version uses local shadowing of the scene data, as discussed in §4. The last version uses a dynamic scheduling implementation of `parallel_for` where the SPE and PPE threads dequeue work from a shared queue, and thereby load balance amongst themselves.

Kernel	B&W Median	Col. Mean	Col. Median	Emboss	Laplacian	Sharpen	Sobel	Greyscale
Speedup	7.7×	7.4×	4.5×	3.6×	3.1×	5.3×	5.7×	3×

Fig. 5. Speedup for Image Kernels

Scene	caustic	caustic3	complex	cornell large	cornell	simple
Global scene data	2.5×	2.6×	1.4×	4.5×	4.4×	2.7×
Local scene data	2.8×	3.0×	7.1×	7.2×	7.1×	3.1×
Dynamic parallel_for	4.9×	5.2×	10.1×	8.9×	8.5×	5.1×

Fig. 6. Speedup for SmallPT Raytracer using `parallel_for`

PARSEC Black-Scholes: Conversion of the Black-Scholes benchmark was straightforward. A single `parallel_for` template function call represents the kernel of the application. We obtained a speedup of 4.0× relative to the serial version on PPE.

PARSEC Swaptions. The code was refactored in two stages. First, dynamic memory allocations were annotated to distinguish between memory spaces. Secondly, unrestricted pointer usage was replaced with static arrays. Local shadowing optimisations were also applied. After these modifications, a 3.0× speedup was obtained.

6 Conclusions

We have shown how, using Offload C++, the TBB `parallel_for` construct can be readily used to distribute work across the SPE and PPE cores of the Cell processor. Our proof of concept implementation provides both static and dynamic work division and supports a subset of the TBB library; `parallel_for` and `parallel_reduce`; the associated `blocked_range` templates, and the `spin_mutex` class object.

We have also demonstrated that data transfer operations can be portably implemented, exploiting target-specific DMA transfer capabilities when instantiated in the context of code to be compiled for the SPE processors.

While related work is available for Cell, the approach of Offload C++ remains distinct. OpenMP targets *homogeneous* shared-memory architectures; although distributed and heterogeneous implementations do exist [8,9]. In contrast to OpenMP on Cell [9], the Offload compiler can use C++ templates to reflect information obtained statically from the call graph, allowing users to optimise code using "specialised" template strategies selected for a specific target architecture *e.g.* the SPE. OpenCL [10] also permits programming in heterogeneous parallel environments. Unlike Offload, OpenCL introduces "boilerplate" code to transfer data between distinct memory spaces via an API, and requires accelerator code to be written in the OpenCL language.

Extending Offload C++ to massively parallel systems, such as GPUs, is likely to follow. However, GPU-like architectures are not an ideal fit for the current Offload C++ programming model, which anticipates random access to a shared global store. Adapting existing application code and Offload C++ to work with the restricted programming models associated with GPUs will be a significant research challenge.

References

1. Hofstee, H.P.: Power efficient processor architecture and the Cell processor. In: HPCA, pp. 258–262. IEEE Computer Society, Los Alamitos (2005)
2. Intel, Threading Building Blocks 3.0 for Open Source, `http://www.opentbb.org`
3. Cooper, P., Dolinsky, U., Donaldson, A., Richards, A., Riley, C., Russell, G.: Offload – automating code migration to heterogeneous multicore systems. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 337–352. Springer, Heidelberg (2010)
4. Codeplay Software Ltd, Offload: Community Edition, `http://offload.codeplay.com`
5. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley, Reading (1994)

6. Bucciarelli, D.: SmallPT-GPU, http://davibu.interfree.it/opencl/smallptgpu/smallptGPU.html
7. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: PACT 2008, pp. 72–81. ACM, New York (2008)
8. Hoeflinger, J.P.: Extending OpenMP to Clusters (2006), http://www.intel.com
9. O'Brien, K., O'Brien, K.M., Sura, Z., Chen, T., Zhang, T.: Supporting OpenMP on Cell. International Journal of Parallel Programming 36(3), 289–311 (2008)
10. Khronos Group, The OpenCL specification, http://www.khronos.org