The Impact of Diverse Memory Architectures on Multicore Consumer Software

An industrial perspective from the video games domain

George Russell Colin Riley
Neil Henning Uwe Dolinsky
Andrew Richards *
Codeplay Software Ltd.
{george,colin,neil,uwe,andrew}@codeplay.com

Alastair F. Donaldson †
Computer Science Department,
University of Oxford
alad@comlab.ox.ac.uk

Alexander S. van Amesfoort ‡
Delft University of Technology
Dept. of Software Technology
a.s.vanamesfoort@tudelft.nl

Abstract

Memory architectures need to adapt in order for performance and scalability to be achieved in software for multicore systems. In this paper, we discuss the impact of techniques for scalable memory architectures, especially the use of multiple, non-cache-coherent memory spaces, on the implementation and performance of consumer software. Primarily, we report extensive real-world experience in this area gained by Codeplay Software Ltd., a software tools company working in the area of compilers for video games and GPU software. We discuss the solutions we use to handle variations in memory architecture in consumer software, and the impact such variations have on software development effort and, consequently, development cost. This paper introduces preliminary findings regarding impact on software, in advance of a larger-scale analysis planned over the next few years. The techniques discussed have been employed successfully in the development and optimisation of a shipping AAA cross-platform video game.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming

* The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n248481 (PEPPHER Project, www.peppher.eu).

† Supported by EPSRC grant EP/G051100/1

‡ Contributions to this work were carried out while the author was an intern at Codeplay Software Ltd., supported in part by the European Union NoE HiPEAC-2 (FP7/ICT 217068)

General Terms Performance, Design, Languages, Experimentation

Keywords memory architecture, performance, multi-core, accessor class, offload

1. Introduction

Video game development drives computer hardware advances, and is an effective method of subsidising or recouping R&D costs. Large AAA[1] titles rival opening weekend and total revenue figures of big hollywood productions. AAA commercial games are large, complex, high performance, real-time software systems targeting multiple hardware platforms. Due to the need for high performance, as well as object-oriented structuring mechanisms, they are almost universally implemented using C++. Codebases are large and evolve rapidly due to changing design requirements. Tight bounds exist on acceptable resource usage, both for processing time and storage capacity.

In response to requests from game companies for tools to support the development of portable games that can run on architectures with multiple, disjoint memory spaces, we have designed Offload C++ [2], a compiler and runtime system for offloading large portions of C++ applications to run on accelerator cores in heterogeneous multicore systems. Since early 2009, we have worked with a number of AAA game providers on applying Offload C++ to their code bases. During these large video game development projects, we have observed that a large proportion of effort is spent attempting to achieve high performance from the memory architecture of the target systems. Development costs are directly related to this effort. This is compounded by the requirement to accommodate highly different memory architectures (*e.g.* the radically different memory architectures of the PlayStation 3 and XBox 360 games consoles) within

[1] In the video games industry, an AAA, or Triple-A title, refers to a high-budget ((tens of) millions of dollars) video game title, with 100+ people actively involved at any point in time for 1–3 years.

portable software, without compromising on performance. Catering to individual memory architectures can present a significant obstacle to code portability, reuse, and future proofing. This is undesirable, given the costs associated with development of software.

Contributions of this paper. After providing an overview of the challenges posed by modern memory architecture and surveying related work (Section 2), we provide a summary of Offload C++ (Section 3). We then discuss the impact of scalable memory architectures on the design of consumer C++ software, describing problems we have faced when applying Offload C++ to large games code bases, outlining techniques we have designed to partially solve these problems, and discussing where these solutions fall short (Section 4). We then discuss an orthogonal issue related to memory systems, that of *indexed addressing* (Section 5), and describe how Codeplay's compiler tools help developers write efficient code for architectures with indexed addressing.

2. Scalable Memory Architectures

Processor performance has outpaced memory performance, leading to the memory wall [11], which limits scalable performance in multicore architectures. Recent memory architectures aim to accomodate multicore scalability by adopting custom features such as explicitly managed, local scratch-pad memory and DMA transfer mechanisms (*e.g.* the Cell BE processor [6], as found in the Sony PlayStation 3 console), or on-chip networks (as employed by Intel's 48-core Single-Chip Cloud Computer (SCC) processor [9]). Scalability of memory systems has also been improved by measures such as dropping formerly standard features like cache-coherency and byte (store) addressing, and by increasing alignment restrictions.

GPU memory architectures are geared towards providing massive bandwidth. This only works well if memory accesses can be combined into large, vector requests that are equally divided over available memory banks. Instead of huge cache hierarchies, in GPU designs, resources are spent on computational and latency tolerating logic. Some fixed-function hardware is present to accelerate texturing (fetching, 2D cache locality and (de)compression). External accelerators like GPUs have the common disadvantage that significant overhead is required to transfer data between host and device memory. Recently, CPU and GPU architectures appear to move in a somewhat more converging direction. CPUs integrate some GPU cores, while memory systems of GPUs start to cache more aggressively.

A software view of memory. The traditional software view of memory is as a single, flat address space with a uniform access cost and large capacity. Highly optimised software may take care to ensure optimal cache behaviour, but details of cache architectures are not exposed directly to software.

Changes to memory architectures are subverting this fundamental assumption made by software, which is in turn reflected in programming languages and programming styles. If a platform has special mechanisms to manage data, such as explicit DMA transfers or other local storage manipulation functions, then at the lowest level these can be exposed via intrinsics. When writing SPE code in high performance games for the PlayStation 3 console, developers are forced to write low-level DMA operations using intrinsics. For example, Figure 1 shows what the programmer might write to pull two game entities involved in a collision pair into local store, update their state according to the collision, and write the results back to main memory. The `dma_get` and `dma_put` operations are non-blocking, and are passed an associated tag `t`, hence the two game entities are fetched in parallel; `dma_wait(t)` blocks until all pending operations associated with tag `t` have completed. Correct synchronization of DMA operations is essential for software correctness, but difficult to achieve in practice. The difficulty of DMA programming has prompted design of both static [3] and dynamic [7] analysis tools to detect DMA races.

```
GameEntity e1, e2; // Allocated in local store

// Fetch game entities associated with collision
dma_get(&e1, collisionPair->first, sizeof(GameEntity), t);
dma_get(&e2, collisionPair->second, sizeof(GameEntity), t);
dma_wait(t); // Block until data arrives

do_collision_response(&e1, &e2);

// Write back updated entities
dma_put(&e1, collisionPair->first, sizeof(GameEntity), t);
dma_put(&e2, collisionPair->second, sizeof(GameEntity), t);
```

Figure 1. Example illustrating the use of explicit DMA for data movement in games code

Memory hierarchy-aware languages and tools. Intrinsic-based programming is low-level and non-portable. Many recent programming languages expose a hierarchical view of memory, enabling programmers to exploit data locality directly, yet attempting to minimise manual effort on the part of the programmer. OpenCL [8] and CUDA [12] expose a three level hierarchy, of 'global', 'local', and 'private' memory to programs, Offload C++ [2] (see Section 3) exposes a two level hierarchy of 'outer' and 'local' memory, while Sequoia [5] exposes a multi-level model.

Alternatively, a compiler or run-time system may provide an application with the illusion of a flat address space, using underlying mechanisms and predetermined policies to move code and data transparently to where in the memory hierarchy it may be most efficiently accessed. This is performed in the XLC C++ and OpenMP compiler [4] and the Hera JVM [10] for the Cell BE.

3. Offload C++

In Section 4, we describe problems we faced when adapting AAA games code to run on systems with multiple memory spaces, using Offload C++. We first give a brief overview of Offload C++; for full details please see [2].

```
void GameWorld::doFrame(...) {
  __offload_handle_t h = __offload {
    // Offload to accelerator
    this->calculateStrategy(...);
  };
  this->detectCollisions(); // Executed in parallel by host
  __offload_join(h); // Wait for accelerator to complete
  this->updateEntities();
  this->renderFrame();
}
```

Figure 2. Simplified example showing how an offload block could be used in a games context

Offload C++ extends the C++ language with a new keyword, `__offload`, which is used to prefix a lexical scope to indicate that the code within the scope should begin executing asynchronously in a separate thread. The lexical scope is called an *offload block*. Offload C++ is geared towards architectures like the Cell BE, consisting of a host core and a number of accelerators, where the accelerator and host instruction sets are different, and where each accelerator is equipped with its own private, scratch-pad memory. In the Cell architecture, the host is the Power Processor Element (PPE), and the accelerators are the Synergistic Processor Elements (SPEs), each of which has 256K of scratch-pad memory. In this context, wrapping a portion of code in an offload block indicates that:

1. The code within the block should be offloaded to an *accelerator* core (SPE)
2. The host should continue to execute, so that the offloaded accelerator thread runs in parallel
3. Data declared inside the offload block should be allocated in scratch-pad memory
4. Access from within the offload block to data declared outside the block should result in automatically generated data-movement code

Making this work for full C++ involves solving some challenging issues related to (1) and (4). For (1), it is necessary to statically identify all code invoked (directly, or indirectly through chains of possibly virtual function calls) from the offload block and compile it separately for the accelerator cores. For (4), to compile pointer dereferences it must be possible to determine whether a pointer refers to local memory, in which case a standard load/store can be issued, or to host memory, in which case an inter-memory space data transfer must be initiated.

Problem (1) is solved by equipping the compiler with techniques for automatic function duplication. There are two cases where manual annotations are required to help the compiler: one is when a call graph rooted in an offload block calls functions in separate compilation units, which are not immediately available for compilation. The other is that the programmer must specify which methods or functions may be called virtually or via function pointer inside an offload block. This list of methods, called a *domain*, and techniques for dynamic dispatch, are discussed in Section 4.1.

Problem (4) is solved via an extended type system. Pointers and references declared inside an offload block scope are automatically type qualified with a new **`__outer`** qualifier if they reside on the accelerator but reference host memory. Offload C++ maintains strong type checking to refuse erroneous pointer manipulations such as assignments between pointers into different memory spaces.

Example. Figure 2 shows a simplified version of a video game loop, where AI computation is performed for game entities (`calculateStrategy`), collisions are detected (`detect Collisions`), and the results of these are used to update and render the game world (`updateEntities` and `renderFrame`). Suppose that strategy calculation and collision detection can be safely performed in parallel, and we wish to offload strategy calculation to an accelerator. This is achieved by wrapping this call in an offload block, by using the `__offload` keyword as shown in the figure. (In practice, some additional syntax is used to pass parameters to the block, which we do not discuss due to space limitations.) When execution reaches the offload block, the Offload C++ runtime launches an accelerator thread to execute `calculateStrategy`. To do so, an accelerator version of this method, *and all methods it may call*, must be pre-compiled. The runtime immediately returns control to the host thread, which executes `detectCollisions` in parallel with the accelerator. The `__offload_join` library function is used to synchronize the offloaded thread, when its results are required. During execution of `calculateStrategy` by the accelerator, any accesses to host memory are automatically compiled into data transfers that go through a software cache.

4. The Impact of Scalable Memory Architectures on Consumer Software

We now discuss the impact of changes to memory architectures on the design of video games, challenges these changes have posed when applying Offload C++ to offload large fragments of game code in systems with multiple memory spaces, and solutions we have used to solve these problems.

Game code is typically structured such that computation is specified as parallel, distinct tasks with well defined synchronisation points executing in a pre-defined and fixed schedule each frame. Tasks perform complex processing on relatively small numbers of objects (100's – 1000's), for purposes ranging from animation, AI, collision detection, physics, and rendering.

4.1 Virtual Methods

Object-oriented design and programming relies heavily on virtual methods, or dynamic dispatch. These are ubiquitous in games code, yet efficiently implementing virtual methods for systems with multiple memory spaces is a challenge.

Consider a C++ method call, `obj->f(...)`. In a traditional system with a single memory space, the 'obj' pointer is dereferenced to obtain a pointer to the virtual table

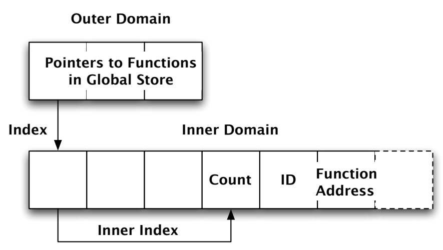

Figure 3. Virtual calls with multiple memory spaces

(*vtable*). The virtual table pointer is dereferenced with an offset to obtain the address for the particular implementation of method `f` to call. Consider now a system equipped with accelerator cores, where each accelerator has a private local memory, and suppose that `obj` refers to an object stored in the local memory of a particular accelerator. Consider the invocation of `obj->f(...)` by a thread running on this accelerator core. It seems natural that we would wish the method call to be executed by the accelerator, operating on fast local memory. Conversely, if `obj` refers to an object in main memory, and the call is issued by the host core, we would expect the appropriate implementation of `f` to be executed by the host. To make this work in practice, for systems where different types of cores have incompatible instruction set architectures, we must move away from the setting where each object type has a single associated vtable.

We now describe the implementation of the system used for virtual method dispatch within Offload C++ for the Cell BE. First, as mentioned in Section 3, the Offload C++ language allows the programmer to annotate offloaded code with details of which functions and methods are expected to be invoked through dynamic dispatch. These will be precompiled to execute on an accelerator core using local memory. When compiling an offload block for an SPE core, the compiler tracks the context of call sites, to determine whether a call requires dynamic dispatch (either because it is a virtual call, or through a function pointer). In this case, a system of inner and outer domains is used, as illustrated in Figure 3. Instead of a normal vtable lookup and call, a domain lookup is performed after vtable lookup to determine if an implementation of the routine is present in the local memory space. This lookup is a two stage process. First, a search over an array of known virtual method addresses, the *outer domain*, determines whether the routine is present in local store. If a potential match is found in the outer domain, the index of the matching pointer in the outer domain is used to index into the *inner domain*. Within the inner domain, we obtain details of function duplicates present – distinct combinations of memory spaces in arguments require distinct duplicates to be made with the appropriate data transfer code. Overloads may be selectively compiled, so there is no guarantee that a full set is present. The inner domain details the number of duplicates present, in a sequence of identifier, function address pairs. The identifier is compiler generated meta-data to identify the signature of the routine with respect to combinations of memory spaces. Once a function address is obtained with an id matching the desired duplicate, the address of the desired routine in local store is known, and the call can be performed.

At present, if a dynamically dispatched function does not provide a match in the inner domain, an exception is generated, providing information which the programmer can use to tell the compiler which methods should be pre-compiled for local dynamic dispatch. Elaborations on this technique could implement alternative behaviours, such as on-demand code loading for functions not present in local memory.

Practical experience offloading virtual methods in games. In the games domain, there is usually only a small selection of virtual functions that can be called at a given point in an algorithm. For example, in physics computations, virtual calls to collision callback methods are invoked (one method for each combination of object types that might collide), while during game AI, specific checks used in decision making involve virtual invocations. Thus annotating a given portion of code to be offloaded with a set of methods to be called virtually is often not too onerous.

However, we have found practical cases where virtual method annotations start to explode. Recently, in applying Offload C++ to a AAA title, we found that, without performing code restructuring, it was necessary to annotate a portion of offloaded code with upwards of 100 virtual functions. The problem was that the game used an abstract component system, performing more than ~ 1300 virtual calls per frame, which we tried to offload in its entirety. We realised that, for a given task, only a selection of types and virtual methods were actually used when the portion of code offloaded was executed. We therefore restructured the component system to be type specialised, in 1 day, and without loss of generality. We wrote a separate offload for each task, one per component, instead of a single offload for all the distinct components, resulting in 13 separate type-specialised offloads. After the restructuring, the maximum number of virtual functions associated with a portion of offloaded code being shipped in this particular game is ~ 40.

For performance critical regions of code, *avoiding* virtual methods is important when offloading to systems with multiple memory spaces. The uniform abstraction of a virtual call such as `move()` hides the specific type, and hence size, of the object on which the function is invoked. Consequently, the object data cannot be prefetched into fast local store. We have found that it is possible to achieve sufficient performance when all commonly accessed data cannot be prefetched. However, processing objects in groups of uniform type permits prefetching and double buffered transfers, for further performance increases. Thus, despite the support for virtual methods in the presence of multiple memory spaces provided by the Offload compiler, developers may still need to spend time rewriting portions of code to avoid virtual calls. Nevertheless, the support provided by Offload C++ provides an important stepping stone between no offloading whatsoever, and an offloaded and fully optimized routine. It took 1 developer 2 months to offload the very complex existing AI code of a AAA game to SPU, with $\sim$ 200 lines of additional code resulting in a 50% performance increase. We note that the restructuring to avoid virtual method calls additionally improved performance across the range of target platforms, including targets with traditional memory architectures.

4.2 Data Locality Optimisations

Data locality and virtual calls. Virtual calls can be expensive when performed repeatedly in a tight loop, with little computation per invocation. Unfortunately, this use case is common in games deigned in an OO style. Consider the loop in the following example:

```
GameObject* objects[N_OBJECTS];
...
GameObject *current = &objects[0];
for (int i=0; i < N_OBJECTS; i++)
    { current->move(); current++; }
```

which iterates over a collection of pointers to objects, invoking virtual method `move` on each. There is a negative performance impact when both the collection of pointers (`objects`), and the referenced objects (targets of pointers in `objects`) are allocated in a non-local memory space where access is via a high latency channel. On each iteration, the current pointer indexing into the container is dereferenced, triggering a transfer between memory spaces, to obtain a pointer to an object, which is dereferenced again to locate a function address in the vtable to perform the virtual call. Each iteration therefore incurs the latency of two dependent memory transfer operations, a significant overhead, especially in the case where the per object computation does not take substantial time.

Software Cache Systems. Cache systems have been implemented in software for diverse memory architectures to mitigate transfer overhead [1, 15]. Software cache lookup introduces some overhead, but this is typically outweighed by the performance increase from avoiding performing repeated accesses to data via inter-memory transfers, a problem common in adapting existing code to support multiple memory space architectures. A compiler can identify and alter each access to use a cache when appropriate during compilation. Offload C++ provides this facility, through its type system which differentiates between pointers to local and host data from within an offload block (see Section 3). To support this, we have developed several software caches, favouring different types of application behaviour. The programmer must decide, based on profiling, which cache is most suitable for a given offload; this is beyond the scope of the compiler.

Accessor Classes and Source Level Portability.

Programmers can use portable accessor classes (efficient data access abstractions) and knowledge of their application's access patterns to achieve high performance. Offload provides a number of accessor classes, written using C++ templates and operator overloading [14].

To minimize transfers, and to better exploit a fast local store, we use an `Array` accessor class as follows:

```
GameObject* objects[N_OBJECTS];
...
Array<GameObject*,N_OBJECTS> local_objects;
GameObject *current = &local_objects[0];
for (int i=0; i < N_OBJECTS; i++)
    { current->move(); current++; }
```

We have interposed an `Array` data accessor between the original array, and the code to access that array. This accessor class is part of the Offload C++ library, and will perform a single, efficient bulk transfer of the array of pointers into fast local store. Subsequently, it acts like an array, allowing indexing operations. This removes a high latency transfer operation from each iteration, increasing performance. On a shared memory system, an `Array` implementation provides direct access to data We have not explicitly stated how the array is to be transferred: this can be factored out in the implementation of `Array`, permitting the use of this technique on portable code.

We have added such data transfer code to optimise offloaded game code compiled using function duplication when executing on multiple address space systems. Similar source changes will arise in systems with multiple memories in single address space, where consideration must be given to inter memory transfers .

5. Indexed Addressing

Memory systems usually are based on addresses that refer to bytes. Some addressing systems, however, are word-oriented (*e.g.* TigerSHARC) or vector-oriented (*e.g.* PlayStation 2 vector unit). In such systems, using an assembler instruction to add 1 to an address causes the address to refer to the next word or vector, instead of the next byte. This allows a much simpler memory architecture, and such a system can be used to index into registers instead of memory. However, this can cause problems with software, as most modern software assumes byte addressing. The problems are only normally visible when defining pointers to data that is smaller than the addressing unit size, or when accessing elements of data structures which are smaller than the addressing unit size.

Solutions to this problem are: use a language that uses word addressing, such as BCPL [13]; keep all pointers as byte-pointers and convert when dereferencing, or use a hybrid approach that allows most existing byte-pointer code to work on word-addressed systems. BCPL uses a system whereby all pointers are word pointers. When processing byte pointers (*e.g.* for strings) special library routines are used. Such languages are rare on modern systems, creating a code portability problem. Keeping pointers as byte-pointers and converting on dereference gives the greatest level of portability, but at the expense of an often unacceptable performance hit. Compiling a pointer dereference (a common operation in C++ code) may require several shifts and some logical operations. The following example works when keeping pointers as byte-pointers, but may be inefficient on word-addressed architectures:

```
for (int i=0; i<N; i++) { *string++ = (char)i; }
```

In designing compilers for game vendors, we have devised a hybrid approach that largely maintains software portability. The novelty of our approach is that the compiler statically generates errors when applied to code that is inefficient for the device. We define an extra attribute for each pointer data type: the addressing unit size. So we can have two types of char pointers:

```
char __word *p; // word-addressed pointer to a char
char __byte *p; // byte-addressed pointer to a char
```

By default, a non-annotated pointer is treated as word-addressed. This means that, by default, any **char** pointer `p` can only point to word-aligned bytes. If we do arithmetic on the pointer, *e.g.* computing `p+1`, then we ensure that the type is now byte-addressed, but the value contains a word-addressed element (in this case `p`) and a byte-addressed offset (in this case 1). This makes the operation of dereferencing the pointer quite efficient: we know that we can load a word at the address pointed to by p, and that we then extract the second byte from that word, which we can compile efficiently, because we know it is a constant value of 1. If we attempt to add an integer variable `x` to `p`, then we know that we will get a variable byte-pointer, and that cannot be dereferenced efficiently, so we raise a compilation error. This forces the user to think about the pointer arithmetic they are performing, and rewrite it in a more efficient manner.

An extended type-checker allows pointer expressions derived from word-addressed pointers to be assigned to byte-addressed pointers, but prohibits non-word-addressed values from being assigned to word-addressed pointers. This is illustrated by the following, where `p` is word-addressed:

```
char *q = p+4; // this is legal, if the word size is 4
char *q = p+1; // this is illegal
char __byte *q = p+1; // this is legal
```

This allows us to define and operate on structures containing byte elements, which is the most common use-case for byte-addressing on word-addressed memory architectures:

```
struct T {
  char a, b, c, d;
} *p;  // this is a word-pointer to data of type T
// This works, using the constant offsets of 'a' and 'b'
p->a = p->b;
```

Our technique does *not* easily support operations on arrays of characters (*i.e.* strings). This is rarely a problem in practice: processors equipped with word-addressed memory are not usually designed to operate on text, hence the decision to use word addressing in the first place. We have found that game developers prefer the hybrid technique when they want to be highlighted of inefficient code generation.

6. Conclusions

The issue of how to improve consumer software, such as games, to make better use of memory architectures is a multi-faceted problem. We have illustrated some problematic hardware / software interactions that occur at present, and suggested techniques for dealing with these instances, with illustrative examples from Offload C++, a programming language and system for offloading large portions of C++ code to run on accelerator cores with local stores.

Exploiting the full performance of architectures with multiple memory spaces in an existing code base can be a complex, costly process. Yet, the alternative of rewriting software from scratch for each new architecture is even less appealing. We believe that this issue has to be attacked from two directions. Software developers require better compilers and portable libraries to support new memory architectures, while common patterns for memory-aware application refactoring must be identified.

References

[1] J. Balart *et al.* A novel asynchronous software cache implementation for the Cell-BE processor. *LCPC*, 2008.

[2] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload - automating code migration to heterogeneous multicore systems. *HiPEAC*, 2010.

[3] A. F. Donaldson, D. Kroening, and P. Ruemmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. *TACAS*, 2010.

[4] A. E. Eichenberger *et al.* Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Systems Journal*, 45:59–84, 2006.

[5] K. Fatahalian *et al.* Sequoia: programming the memory hierarchy. *SC*, 2006.

[6] H. P. Hofstee. Power efficient processor architecture and the Cell processor. *HPCA*, 2005.

[7] IBM. *Cell BE Race Check Library*, July 2008. Described in Example Library API Reference, version 3.1.

[8] Khronos Group. The OpenCL specification. `http://www.khronos.org/opencl`.

[9] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmer's view. *SC*, 2010.

[10] R. McIlroy and J. Sventek. Hera-JVM: a runtime system for heterogeneous multi-core architectures. *OOPSLA*, 2010.

[11] S. A. McKee. Reflections on the memory wall. *Computing Frontiers*, 2004.

[12] NVIDIA. CUDA zone. http://www.nvidia.com/cuda/.

[13] M. Richards. BCPL: a tool for compiler writing and system programming. *AFIPS Spring Joint Computer Conference*, 1969.

[14] G. Russell, P. Keir, A. Donaldson, U. Dolinsky, A. Richards, and C. Riley. Programming heterogeneous multicore systems using threading building blocks. *HPPC*, 2010.

[15] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. *ICS*, 2008.