

# CUBA: An Architecture for Efficient CPU/Co-processor Data Communication

Isaac Gelado\*, John H. Kelm†, Shane Ryoo†,  
Steven S. Lumetta†, Nacho Navarro\* and Wen-mei W. Hwu†

\*Departament d'Arquitectura de Computadors  
Universitat Politècnica de Catalunya  
Barcelona, Spain  
{igelado,nacho}@ac.upc.edu

†Center for Reliable and High-Performance Computing  
University of Illinois at Urbana-Champaign  
Illinois, United States  
{jkelm2,sryoo,steve,hwu}@crhc.uiuc.edu

## ABSTRACT

Data-parallel co-processors have the potential to improve performance in highly parallel regions of code when coupled to a general-purpose CPU. However, applications often have to be modified in non-intuitive and complicated ways to mitigate the cost of data marshalling between the CPU and the co-processor. In some applications the overheads cannot be amortized and co-processors are unable to provide benefit. The additional effort and complexity of incorporating co-processors makes it difficult, if not impossible, to effectively utilize co-processors in large applications.

This paper presents CUBA, an architecture model where co-processors encapsulated as function calls can efficiently access their input and output data structures through pointer parameters. The key idea is to map the data structures required by the co-processor to the co-processor local memory as opposed to the CPU's main memory. The mapping in CUBA preserves the original layout of the shared data structures hosted in the co-processor local memory. The mapping renders the data marshalling process unnecessary and reduces the need for code changes in order to use the co-processors. CUBA allows the CPU to cache hosted data structures with a selective write-through cache policy, allowing the CPU to access hosted data structures while supporting efficient communication with the co-processors. Benchmark simulation results show that a CUBA-based system can approach optimal transfer rates while requiring few changes to the code that executes on the CPU.

## Categories and Subject Descriptors

C.0 [Computer Systems Organization]: General

## General Terms

Design

## 1. Introduction

CUBA (Champaign-Urbana/Barcelona) is an architecture model for coupling data-parallel co-processors to general-purpose CPUs. CUBA incorporates mechanisms for reduc-

ing the communication latency and data marshalling overheads incurred when moving data between CPUs and co-processors. Lowering the cost of accessing highly data-parallel co-processors, both in terms of prolonged execution time and programming efforts, CUBA allows for a wider range of applications to benefit from data-parallel co-processors.

Data parallelism (DP) refers to the property of an application to have a large number of independent arithmetic operations that can be executed concurrently on different parts of the data set. Data parallelism exists in many important applications that model reality, such as physics simulation, weather prediction, financial analysis, medical imaging, and media processing. We refer to data-parallel phases of an application as *kernels*. Contemporary high-performance CPUs employ instruction-level parallelism (ILP), Single-Instruction-Multiple-Data (SIMD), memory-level parallelism (MLP), and thread-level parallelism (TLP) techniques that all exploit data parallelism to a certain degree. However, due to cost and performance constraints imposed by sequential applications, these CPUs can only dedicate small portions of their resources to the exploitation of data parallelism, which motivates the design of co-processors for exploiting massive amounts of data parallelism.

We define a co-processor as a programmable set of functional units, possibly with its own instruction memory, that is under the control of a general-purpose processor. Some co-processors implement fine-grained computations, such as floating-point arithmetic, vector operations or SIMD instructions. These fine-grained co-processors are integrated into CPUs as functional units that support new processor instructions (e.g., SIMD instructions). Other co-processors, such as the Synergistic Processing Elements (SPEs) in the Cell BE processor [5], the NVIDIA GeForce 8 Series graphics processing units, Physics Engines or the reconfigurable logic in the Cray XD1 [4], execute medium- and coarse-grained operations. The software abstraction used by many coarse-grained co-processors today is a threaded execution model such as SPU threads in the Cell SDK.

In this paper we focus on coarse-grained co-processors that can potentially exploit a large amount of data parallelism. We argue that a programming model where coarse-grained co-processors are *encapsulated as function calls* is a useful and powerful model that presents two main benefits. First, function call encapsulation provides co-processors with a simple model similar to libraries and API (Application Programming Interface) functions, which is familiar to developers. As a result, co-processors can be easily incorpo-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece  
Copyright 2008 ACM 978-60558-158-3/08/06 ...\$5.00.

```

int dinst1(uchar *blk1, uchar *blk2, int len)
{
    int i,j,s;
    uchar *a,*b;
    s=0; a=blk1; b=blk2;
    for(j=0;j<16;j++) {
        for(i=0;i<16;i++) {
            s+=abs(a[i]-b[i]);
        }
        a+=len;
        b+=len;
    }
    return(s);
}

```

(a) Motion Estimation, pixel distance function

```

int fullsearch(uchar *org, uchar *blk, int len,
              int i0, int j0, int win)
{
    int m,k,d,i,j;
    . . .
    for(m=1;m<=win;m++) {
        i=i0-m; j=j0-m;
        for(k=0;k<8*m;k++) {
            d=dinst1(org+i+len*j,blk,len);
            . . .
            if(k<2*m) i++;
            else if(k<4*m) j++;
            else if(k<6*m) i--;
            else j--;
        }
    }
}

```

(b) Motion Estimation, full search (spiral pattern)

Figure 1: Source code for motion estimation in MPEG2

rated into current applications with little change to the code. Programmers can use a sequential programming model to develop new applications that take advantage of the parallel processing capabilities of co-processors. Second, CUBA does not prevent developers from making more complex code changes, such as using multiple threads of execution, to improve performance and enable co-processors to be employed when the function call abstraction is not an appropriate option.

Many coarse-grained co-processors are equipped with their own main memory, which we refer to as *local storage*. We discuss two main issues regarding co-processors: communication latency and cost of extracting and arranging data to be sent to the co-processor, the latter of which we refer to as *marshalling*. Current CPU/co-processor architectures are accessed using a long-latency system bus. The CUBA architecture requires local storage for co-processors and uses a mechanism to collect data produced by the CPU providing it to the co-processor local storage in the same layout as the original CPU layout, obviating the need for the application to do explicit marshalling. Reducing the overhead associated with communication and marshalling allows CUBA to provide a co-processor architecture that can better accelerate a wider array of data-parallel kernels.

We evaluate CUBA using multiple benchmarks on a cycle-accurate simulator of a superscalar processor attached to a data-parallel co-processor. CUBA is shown to reduce the programming efforts and run-time communication overhead for accessing co-processors compared to alternative designs. We demonstrate speedup across our benchmarks with respect to previous DMA-based CPU/co-processor architectures, which have already shown speedups from 10X to 240X with respect to a baseline CPU-only configuration [18].

The main contributions of this paper are: (1) A programming model that encapsulates co-processors as function calls, which provides a simple programming model for using data-parallel co-processors coupled to general-purpose CPUs. (2) An analysis of data access and marshalling overheads incurred when using traditional co-processors that implement data-parallel computations. Our analysis demonstrates the importance of reducing these overheads and has implications for the programming models used to access co-processor functionality. (3) A mechanism to map program data in the co-processor local memory. The main benefit of our mechanism is that it does not penalize CPU accesses to data not hosted by the co-processor local storage and requires little additional logic. (4) A description of the hard-

ware support a co-processor would implement to allow the virtualization of the co-processor local storage.

This paper is organized as follows. We present the programming model we adopt for co-processors in Section 2. In Section 3 we describe the modifications to current processors required by CUBA and sketch how to virtualize the co-processor local storage. We describe our experimental methodology in Section 4 and the experimental results in Section 5. In Section 6 we summarize the related work in CPU/co-processor architectures. In Section 7 we conclude and present potential future work.

## 2. Co-processor Programming Model

We present an example of the data parallelism that exists in many applications. Figure 1 shows two functions that implement motion estimation in an MPEG2 video encoder. Function `dinst1()` calculates the sum of absolute differences (SAD) for a block of 256 image pixels. The kernel is rich in data parallelism and amenable to execution by a co-processor designed to exploit a large amount of data parallelism. We will use motion estimation as an example to illustrate the overall programming model adopted in CUBA.

In the programming model adopted for CUBA, each co-processor is viewed as an extension of the CPU and can be invoked in much the same way as a library function. For instance, consider a co-processor that implements the `dinst1()` code in Figure 1a. The call to the co-processor appears in the motion estimation `fullsearch()` code (Figure 1b) as a function call to `dinst1()` as if it were implemented in software. However, the call to `dinst1()` invokes the co-processor instead. The co-processor uses a large number of execution units and potentially a large number of threads to exploit the data-parallelism in the kernel. We will assume that the code running in the CPU calls the co-processor using a special user-mode instruction and synchronizes with the co-processor using a polling loop. Interface logic hides the low-level details of data transfers to and from the co-processors and does not require major modifications in the application code that invokes them. Note that the application could include two versions of the same function: one using the co-processor and another one only using the CPU. Hence, the same application can be executed in systems with and without co-processors. Only co-processor execution is considered in this work.

We have identified three models of data transfer between the CPU and co-processor. The first model is the simplest among the three, but achieves the lowest performance. The

```

int dinst1(uchar *blk1, uchar *blk2, int len)
{
    int i,s;
    uchar *dma_blk1, *dma_blk2;
    dma_blk1=dmalloc(16*16);
    dma_blk2=dmalloc(16*16);
    for(i=0;i<16;i++) {
        dmacpy(dma_blk1+i*16,blk1+i*len,16);
        dmacpy(dma_blk2+i*16,blk2+i*len,16);
    }
    s=dinst_coproc(dma_blk1,dma_blk2,len);
    return(s);
}
(a) Motion Estimation, pixel distance stub
in Per-Call model

int dinst1(uchar *blk1, uchar *blk2, int len)
{
    int s;
    s=dinst_coproc(co_addr(blk1),
                  co_addr(blk2),len);
    return(s);
}
(b) Motion Estimation, pixel distance
stub in Co-Processor-Hosted model

int fullsearch(uchar *org, uchar *blk, int len, int i0, int j0, int win)
{
    int m,k,d,n, q = 0;
    uchar *dma_org[2], *dma_blk;
    . . .
    dma_blk=dmalloc(16*16);
    dma_org[0]=dmalloc(16*16);
    dma_org[1]=dmalloc(16*16);
    for(n=0;n<16;n++) {
        dmacpy(dma_blk+n*16,blk+n*len,16);
        dmacpy(dma_org[0]+n*16,org+n*len,16);
    }
    for(m=1;m<=win;m++) {
        i=i0-m; j=j0-m;
        for(k=0;k<8*m;k++) {
            d=dinst1_coproc(dma_org[q%2],dma_blk,len);
            . . .
            if(k<2*m) i++;
            else if(k<4*m) j++;
            else if(k<6*m) i--;
            else j--;
            q++;
            for(n=0;n<16;n++) dmacpy(dma_org[q%2]+n*16,org+i*len*(n+j),16);
            wait_coproc();
        }
    }
}
(c) Motion Estimation, fullsearch in Double-Buffered model

```

Figure 2: Source code for different co-processor programming models

second model achieves higher performance than the first model, but requires high level of programming effort. The third model is the one we advocate in this paper. It uses a hardware mechanism to achieve higher performance while keeping the additional programming efforts low.

The first data transfer model, *Per-Call*, uses the co-processor local storage to hold only the data required for a single call to the co-processor. This model has the smallest local storage requirement among the three models. In the MPEG2 example, the co-processor local storage holds two  $16 \times 16$  blocks (`blk1[]` and `blk2[]`). The `dinst1()` function on the CPU is modified to collect the relevant subarrays of the `org` and `blk` arrays, then transfers them from main memory to the co-processor local storage before calling the co-processor. In the Per-Call model the complexity of the co-processor data transfer is encapsulated in the `dinst1()` function and the code in `fullsearch()` does not change.

Figure 2a shows the code that collects and transfers the data. The code marshals the input parameters: it collects non-consecutive portions of `org` and `blk` from the CPU memory and transfers the data to consecutive memory locations of the co-processor local storage. The need for data marshalling arises from the fact that the call to the original `dinst1()` function uses the call-by-reference convention for passing the pointers into two large arrays to the function, which uses these pointers to access the subarrays in the memory space. As shown in Figure 1a, after processing each row of the subarrays, the original CPU code advances both pointers `a` and `b` by the row length of the large array. In CPU execution, whenever the two subarrays reside in cache, the `dinst1()` function can access them at very little cost. In the Per-Call model, the explicit collection and transfer of these subarrays to the co-processor local memory effectively changes the calling convention to call-by-value for the subarrays. The data marshalling and transfer times are added to the computation latency for the co-processor, possibly negating the benefit of using a data-parallel co-processor.

The second model, *Double-Buffered*, uses the co-processor local storage to hold the input data required for both the

current and next co-processor calls. The Double-Buffered model may require up to twice the local storage capacity as the Per-Call model, but allows the CPU to transfer data for the next invocation while the co-processor is executing the current call. If the co-processor computation latency is longer than the data transfer latency, the latter can be masked. Furthermore, if some inputs are constant across calls, they can be transferred once and used by many calls. In our MPEG2 example, `blk2[]` is persistent state that is maintained across calls to the `dinst1()` function. If we double buffer `blk1[]`, the co-processor local storage hosts one  $16 \times 16$  block `blk2[]` and two  $16 \times 16$  blocks `blk1[]` for both the current invocation of `dinst1()` and the next invocation.

The code in `fullsearch()` requires major changes to implement double buffering as shown in Figure 2c. Before entering the outer loop, the code collects and transfers `blk1[]` and `blk2[]` to the co-processor local storage. The code inside the inner loop calls `dinst1()`, which returns immediately without returning any value. Immediately after the call, we add code to collect and transfer `blk1[]` for the next invocation and wait for the co-processor to finish. When the co-processor returns the output value, the code in the CPU uses it and iterates in the loop. If the data transfer time takes longer than the computation in the co-processor, the communication overhead becomes unacceptable.

For the Double-Buffered model, the code in `fullsearch()` needs to know the data access range for `dinst1()` in order to collect all elements of `blk1[]`. This requirement violates commonly held software engineering principles and is complicated by the fact that the next block to be processed is determined by a fairly complex set of conditions. There also may be cases where the access range for the next iteration is not determinable at compile time, which makes double buffering even more difficult to implement. Furthermore, the code modifications to `fullsearch()` also depend on the co-processor being used. Hence, the resultant `fullsearch()` code is unlikely to be shared across systems without co-processors or with different co-processors.

The third model is *Co-Processor-Hosted* and is the model adopted for CUBA. In this model the co-processor local storage hosts all data used in a group of consecutive calls to the co-processor by `fullsearch()`, which requires more local storage capacity than the previous models. However, the program data layout in this model can be exactly the same as the data layout for the original application. Hence, there is no data marshalling overhead or major changes to the CPU portion of the application. For the MPEG2 example, the two large arrays, `blk[]` and `org[]`, are mapped to the local storage of the co-processor rather than the CPU main memory. Thus, they are automatically written to the co-processor local storage as soon as they are produced, before the first call to `fullsearch()` (in this example, when read from the disk). Therefore, the data arrives at the co-processor local storage before it is needed by the `dinst1()` function implemented in the co-processor. Figure 2b shows the code inside `dinst1()`. The code calls the co-processor and waits for it to return the output value to `fullsearch()`, which requires no modification to the code in `fullsearch()`. For these reasons, the Co-Processor-Hosted model can be much easier to use for complex applications.

A potential penalty imposed by the Co-Processor-Hosted model is additional access time overhead for the CPU to access these two arrays, which the CUBA architecture addresses by allowing the CPU to cache the contents of the co-processor’s local storage, as discussed in the next section. The other main disadvantage is that it requires more co-processor local storage than the other two models. Assuming a nominal  $640 * 360$  frame size for the MPEG-2 example, the co-processor must host 450KB, instead of the 512B and 768B required for the two previous models, respectively. The local storage capacity requirement can be easily met by off-chip co-processors today (e.g., the NVIDIA 8800 GTX card has 768MB of local storage), but it restricts the set of computations that can be implemented by current on-chip co-processors (e.g., each SPE in the Cell BE has 256KB of local storage). In Section 3 we discuss how to virtualize the co-processor local storage, which enables a given co-processor to support computations that require more memory than its physical local storage.

### 3. Overall Design of CUBA

CUBA is applicable to both on-chip and off-chip co-processors. We will illustrate our descriptions with an off-chip co-processor coupled to a CPU with an interconnect similar to what the *Torrenza* initiative defines [10] (see Figure 3). The CPU and the co-processor are interconnected using a HyperTransport-like link [2]. We assume that the link supports CPU access to the co-processor memory-mapped registers and to the co-processor local storage.

The co-processor, on the other hand, has no direct access to the CPU memory; all its effects on the CPU memory system are through its local storage and managed by a mechanism located in the main memory controller. A benefit of this design is that a CUBA co-processor does not have the ability to accidentally or maliciously corrupt the memory contents of any CPU processes other than the user process calling the co-processor. Thus, a failing co-processor will behave like a defective third party software library function with limited scope of damage.

In this section we will describe the modifications to the CPU memory management unit (MMU) logic and the mem-

ory controller. The objective of these modifications is to selectively map application data structures into the co-processor local storage on the fly to implement the Co-Processor-Hosted model described in the previous section. The application requests this mapping by calling API functions supported by the operating system (details to follow).

#### 3.1 CUBA Architecture Model

In CUBA, the application data structures accessed by co-processors are hosted by the co-processor local storage instead of the CPU main memory. When the CPU reads or writes such data, it effectively accesses the co-processor local storage. The co-processor local storage contents can be cached by the CPU. Thus, CPU accesses to contiguous memory locations or repeated accesses to the same data can take advantage of the short cache latency. For this work, the co-processor local storage is restricted to only contain data for a single application at any time.

CUBA hides the data movement overhead by dynamically collecting the co-processor input data as it is being produced by the CPU. A *Co-processor Local Storage Collector* (CLSC) is integrated with the CPU main memory controller. The CLSC inspects every memory request going into and out of the controller to identify writes to the data that are being hosted by the co-processor. The CLSC determines if an access is for the local storage based on the mapping designated by the application. Whenever a memory write involves the co-processor local storage, the CLSC forces a write-through action from the main memory controller to the co-processor. With additional buffering, the CLSC coalesces writes to sequential locations for better transfer efficiency. The CLSC and the write-through policy for the hosted data structure when they reside in the L1 and L2 caches maximize the probability that when the co-processor starts computing, all of its input data will be already present in its local storage. We will relax the constraint of using a write-through L2 cache later in this paper.

Let us consider an MPEG2 application that uses a co-processor to implement both functions of the motion estimation code in Figure 1. The application first allocates co-processor local storage to store the current and previous video frame objects by calling the operating system, which sets up the necessary mappings. At the end of each time step, the MPEG2 application logically copies the current frame into the previous frame and reads in the new current frame. In the original code, this copying is accomplished by assigning the address of the current frame object of the ending time step to `org` and the address of the new input frame object to `blk` before reading the new input frame.

While the MPEG2 application code reads a new input frame from disk, the CLSC sees a series of memory writes to locations mapped to the co-processor storage. It collects these writes, coalesces them into larger transfer units, and sends them to the co-processor. When the application calls the `fullsearch()` function, it first writes the input parameters (`*org`, `*blk`, `len`, `i0`, `j0` and `win`) into the co-processor registers. The previous and current frame objects are passed by-reference using the `org` and `blk` pointers. When the co-processor finishes computing, the CPU gets the output value from one of the co-processor registers. For the next time step, the application calls `fullsearch()` using the same input parameters, except for `*org` and `*blk`, which contains new addresses in them. Note that `*org` and `*blk` contain ad-

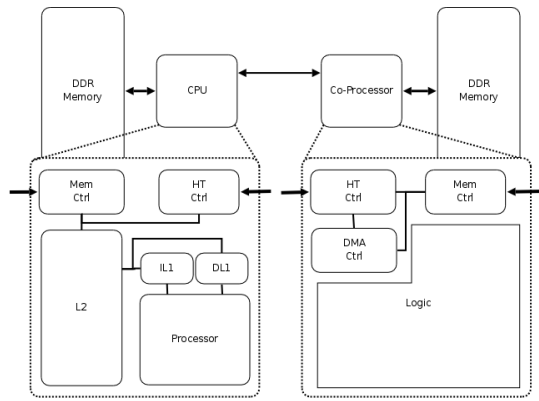


Figure 3: General Model of the CUBA architecture

addresses within the virtual address space of the application. However, when calling the co-processor, the co-processor requires addresses from its own local address space. Thus, when passing these parameters, the application uses the information contained in an OS-provided table to translate from its virtual address space to the co-processor local address space<sup>1</sup>. Finally, the application calls the co-processor to execute the `fullsearch` and `dinst1()` functions.

### 3.2 Co-processor Local Storage Collector

Co-processor data hosting is implemented using a TLB assisted mechanism: the page table entry is extended with a one-bit C field, which is set for those pages containing data hosted by the co-processor. The L1 and L2 caches also store the C field for each cache line.

The C field is used to implement a hybrid write-back/write-through L2 cache. The L2 cache controller follows a write-through/write-no-allocate policy only for those cache lines whose C field is set. A hybrid policy provides three main benefits: (1) the data will be present in the co-processor local storage before launching the computation, (2) co-processor data is still cached so repeated CPU accesses or accesses to contiguous co-processor data are not penalized, and (3) write accesses to other data structures do not incur write-through actions by the L2 cache.

The L2 cache controller sends the C field to the main memory controller for every memory write operation. Write requests whose C bit is set are handled by the CLSC. The CLSC implements a write buffer to store pending write operations to the co-processor local storage. Pending requests in the write buffer are coalesced and sent to the co-processor local storage through the Hyper-Transport link. A single core system requires a write buffer with as many entries as the maximum number of outstanding memory requests the L2 can support. If the ISA does not allow write reordering, coalescing write operations might lead to race conditions. For instance, a write to a lock variable might be done before actually writing the data that is locked. We expect the co-processor local storage to only host “pure data” structures while synchronization variables are stored in main memory.

The CLSC presents the following interface to the CPU: applications use the `scop` (*Start Co-Processor*) instruction to signal the CLSC to complete the data movement neces-

<sup>1</sup> Accessing the table does not require any system call to be performed by the applications.

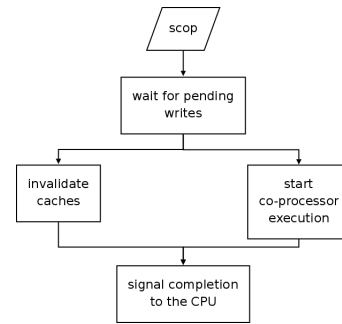


Figure 4: Actions performed in the CLSC when a scop instruction commits

sary and begin execution of the co-processor. Figure 4 shows the actions performed by the CLSC when it gets a `scop` request. First, it waits for all pending write-through activities to the co-processor to finish. Then the CLSC invalidates those cache lines in the L1 and L2 caches whose C field is set and sends the `scop` request to the co-processor. Note that these two actions can be performed in parallel if the CPU does not request any data during this process. To enforce this constraint, when the `scop` instruction enters into the processor issue queue, the processor stops fetching new instructions. When the `scop` instruction commits, it is sent to the CLSC. At this point there are no other instructions in flight since the issue queue is empty. The CLSC signals completion (all write-through activities and invalidations are done) to the CPU by asserting a line that enables fetching and issuing new instructions.

The write-through cache policy for the co-processor data allows us to overlap the execution in the CPU with the data movement, as illustrated in Figure 5a. When the application calls the co-processor, the CLSC ensures that the single valid copy for the co-processor data is in the co-processor local memory (Figure 5b). The application implements a polling loop to wait for the co-processor to finish. Hence, only the co-processor can access those data (Figure 5c) while it is computing. Once the co-processor is done, any CPU access to the co-processor data will miss in the L1 and L2 caches since all the cache lines containing data hosted by the co-processor were invalidated. Both caches fill from the co-processor local storage (Figure 5a).

### 3.3 Benefits and Limitations of CUBA

The design of CUBA has several implications for applications performance. The hybrid write-back/write-through L2 cache used in CUBA is the key to overlap data transfers between the CPU and the co-processor. Furthermore, the hybrid policy increases the probability of having the data present in the co-processor local storage when the CPU calls the co-processor. However, there is a potential side-effect that might reduce the benefit of the hybrid write policy of CUBA. Every time the CPU modifies the data hosted by the co-processor local storage, a write-through action is triggered at the L2 cache. Hence, repeated writes to the same memory location from the CPU before the co-processor is called lead to several write-through actions, increasing the amount of data transferred from the CPU to the co-processor local storage. Thus, the average bandwidth of the L2 cache, the L2 bus, the memory controller and, the Hyper-

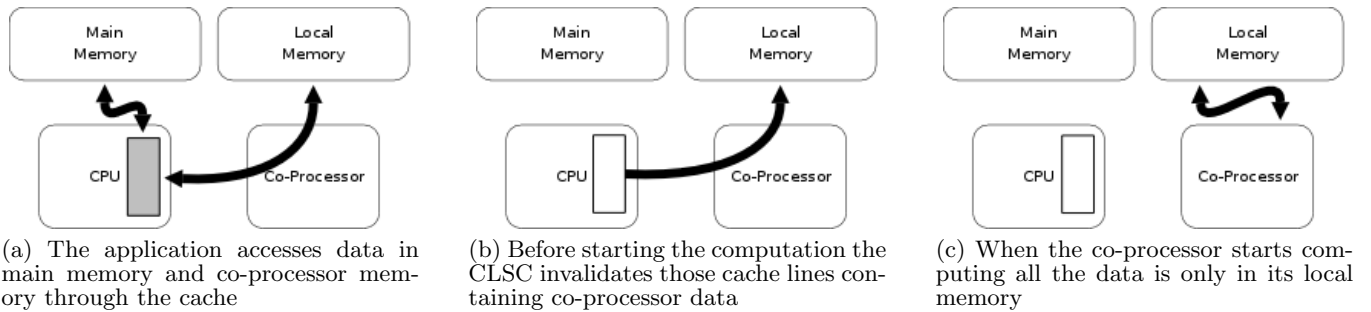


Figure 5: Data movement in CUBA

Transport link required by CUBA is higher than the bandwidth required by traditional DMA architectures. However, CUBA redistributes write and reads operations to the co-processor local storage across the total execution time of the application. Thus, in many cases the maximum instantaneous bandwidth required by CUBA is much smaller than the instantaneous bandwidth required during a DMA transfer between the CPU main memory and the co-processor local storage. A given hardware element only limits applications performance when it is not able to deliver the instantaneous bandwidth required by the application. Because CUBA reduces the maximum value of the instantaneous bandwidth requirements, we expect CUBA to perform better than current DMA-based architectures.

A benefit of CUBA is its ability to perform DMA transfers from I/O devices to the co-processor local storage, since this is mapped in the system physical address space. In many applications, the data used by co-processors is directly read from the disk. In these applications CUBA might decrease the amount of data transferred between the CPU main memory and the co-processor local storage significantly.

### 3.4 Support for Virtualization

CUBA targets a programming model where the co-processor local storage hosts the data used during consecutive invocations of the co-processor (as discussed in Section 2). We expect programmers to map to the co-processor local storage any data structure that might be needed by the co-processor. Hence, some local storage capacity might be wasted. For instance, the programmer might map an array of structures where only one field of many is required by the co-processor.

The capacity of the local storage might limit the number of applications that benefit from CUBA. To overcome this limitation, the co-processor local storage can be virtualized. The virtualization mechanism is used whenever the size of the mapped data overflows the capacity of the local storage. If the overhead produced by the local storage virtualization is unacceptable, programmers might need to re-organize data structures to fit the data in the co-processor local storage. We describe the hardware support necessary to virtualize the co-processor local storage as follows.

Applications request co-processor local memory from the OS, which sets the corresponding entries in the page table to map a co-processor memory range to the application virtual address space. It also sets the C bit for this range. If there is no co-processor memory available, the OS maps a range of main memory instead. When the application executes a `scop`, the co-processor data is split between main memory and co-processor memory.

To virtualize the co-processor local storage, it is necessary to detect memory accesses from the co-processor to data not present in its local storage and send an exception (*data not present*) to be handled by the OS. The OS then selects a range within the co-processor local storage to be swapped out to main memory, swaps in the required data from main memory to the co-processor local storage and modifies the necessary entries in the co-processor page table to reflect the changes in the mappings. The OS code that swaps the data in and out of the co-processor memory must not bring any co-processor data to the L2 cache. This constraint can be easily fulfilled by moving all the data using DMA transfers. First, the OS swaps data out using a DMA transfer from the co-processor local memory to main memory. Then the OS initiates a new DMA transfer to swap the required data from main memory to the co-processor local memory.

The co-processor might send a *data not present* exception while the CLSC is invalidating the caches. This case does not present any problem, because the code that handles the exception will not be executed until the CLSC enables the instruction issue logic again. Therefore, when the processor starts executing the exception handling code, all cache lines have been already invalidated.

## 4. Experimental Setup

We use execution-driven simulation to evaluate the CUBA architecture design. The architecture of the simulated processor is shown in Table 1. We simulate three configurations: *dma-opt*, *dma*, and *cuba*. In *dma-opt*, data transfer is assumed to take zero time and demonstrates the maximum speedup a co-processor system can achieve using DMA. *Dma* uses DMA transfers, similar to what is used by the NVIDIA CUDA model, between the CPU and the co-processor. The simulated DMA controller can read data from the L2 cache if present and from main memory otherwise. DMA transfers use burst reads and writes of 128 bytes.

The evaluation is performed using a cycle-accurate simulator based on SESC [17], which was modified to incorporate DMA transfers, co-processor execution and CUBA.

### 4.1 Workloads

We have selected four different applications suitable for acceleration using massively data parallel co-processors. These applications have been selected from the set of benchmarks presented in [18], where actual speed-ups are reported when using NVIDIA CUDA. We select benchmarks that exhibit different data access patterns and that we are able to simulate in a reasonable time. In our simulations we assume that

Processor	Memory Subsystem
Freq: 5GHz	L1 ICache: 16K (4-way, 1 port)
Fet/Iss/Ret width: 4/4/5	L1 DCache:16K (4-way, 1 port)
LdSt/Int/FP units: 4/4/3	L1 Hit/Miss Delay: 2/13
RAS: 32	L2 Cache: 1MB (16-way, 2 ports)
BTB: 2K entries, 2-way assoc.	L2 Hit/Miss Delay: 10/105
Branch pred:	ITLB entries: 64 (4-way)
bimodal size: 16K entries	DTLB ent: 64 (4-way)
gshare-11 size: 16K entries	Main Mem: 240
I-window: 92	Coproc Mem Delay: 240
ROB size: 176	Coproc Link Channels: 4
Int regs: 96	Coproc Link Latency: 10
FP regs: 80	CLSC WBuff: 32 * 4
Ld/St Q entries: 56/56	
IMSHR/DMSHR: 4/16	

**Table 1: Simulation parameters. Latencies shown in processor cycles representing minimum values.**

Benchmark	CUDA			Kernel Calls
	Co-Proc Execution	Kernel Speed-up	Appl. Speed-up	
h264	2.6%	20.2X	1.47X	2
lbm	98.3%	12.5X	12.3X	20
mri	99%	316X	263X	1
saxpy	88%	19.4X	11.8X	≈26,000

**Table 2: Workloads used in our experiments**

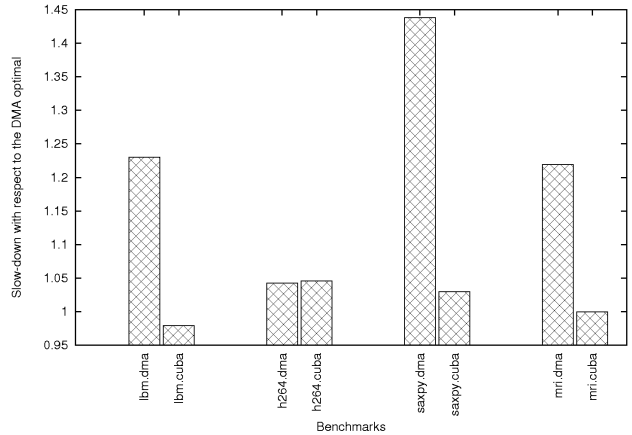
the co-processor achieves the same level of speedup for each application as that reported in [18]. The simulated processor is more aggressive than the one used in [18]; we assume that the co-processor is scaled in its parallelism to match the increased CPU speed. Note that we aim to simulate a generic co-processor, not only GPUs. Thus, the estimated value for the co-processor execution time is likely to fit within the range of possible execution times for a certain co-processor. Table 2 show the benchmarks used in our simulations.

*h264* is based on the 464.h264ref SPEC CPU2006 benchmark, which is an H.264 video encoder. A dependence between motion estimation of consecutive macroblocks in a video frame was removed to enable parallel execution of the motion estimation code on multiple macroblocks. This modification changes the seed motion vector for each macroblock and therefore slightly affects the output of the program, but it is allowed by the H.264 standard. *Lbm* is based on the 470.lbm benchmark from SPEC CPU2006, which uses the Lattice-Boltzmann Method for simulating 3D fluid dynamics. The program has been modified to print fewer status reports. *Mri* implements the computation of a vector  $F^H d$ , an image-specific vector, used in a 3D magnetic resonance image reconstruction algorithm based on non-uniform Fourier Transform. *Saxpy* is a single-precision floating-point implementation of *saxpy* from High-Performance LINPACK, used by a Gaussian elimination routine.

We compile the CPU code using gcc version 3.4 with the `-O3` flag to MIPS binaries. To compare the performance we use the slow-down of the configuration with respect to the *dma-opt* version of the benchmark.

## 4.2 Comparing Data Transfer Mechanisms

We evaluate two different data transfer mechanisms: DMA (*dma*) and CUBA (*cuba*). For the DMA configuration we modify the NVIDIA CUDA version of the benchmark, which has already been optimized to reduce the overhead of data movement between the CPU and the co-processor. The CUDA API calls for allocating DMA memory buffers, per-



**Figure 6: Slow-down with respect to the dma-opt configuration achieved for the benchmarks shown in Table 2 (the lower the better)**

forming data transfers and launching co-processor computation are substituted by the analogous ones provided by the simulation platform. For *dma-opt* we use the same code as the DMA model, but we comment out the code that allocates DMA memory and performs the data transfer. For the *cuba* configuration, we only perform two modifications from the original code: First, we add the necessary calls to allocate the data structures used by the co-processor in the co-processor memory. Second, we substitute the code now executed by the co-processor with the API calls to launch the computation.

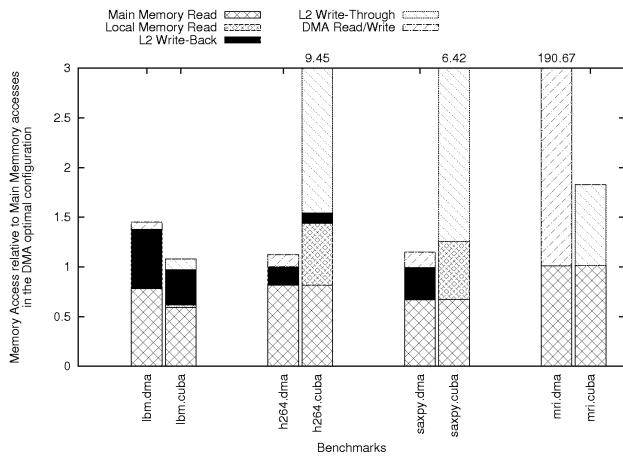
## 5. Evaluation

### 5.1 Performance

We compare the performance of CUBA with DMA data transfer between CPU and Co-processor. Figure 6 shows the results for the different configurations and benchmarks. Results are shown as the the slow-down with respect to the *dma-opt* configuration (instantaneous DMA).

CUBA performs better than the other configurations for three of the four benchmarks we simulate: *lbm*, *mri* and *saxpy*, but for *h264* CUBA is 0.3% slower than DMA. The data transfer between the CPU and the co-processor in all benchmarks benefits from using CUBA. Figure 7 shows that in *h264* there are more writes to the co-processor local memory than in DMA. A detailed analysis of the experimental results shows that the slight slow-down in CUBA is due to a 10.32% increase in the number of TLB misses. In CUBA there are more TLB misses due to the page table entries needed for mapping the co-processor local memory data.

There is an anomaly in *lbm*: CUBA performs better than the DMA optimal configuration. In the DMA configuration four kernels are implemented by the co-processor, but two of these kernels are more efficiently executed by the CPU. *Dma* and *dma-opt* execute them on the co-processor to avoid data movement overheads, but CUBA executes them in the CPU, since the data movement overheads are much smaller. Figure 7 shows that in this benchmark CUBA reduces the number of write-back operations in the L2 cache. In DMA many L2 write-back operations affect the data used by the

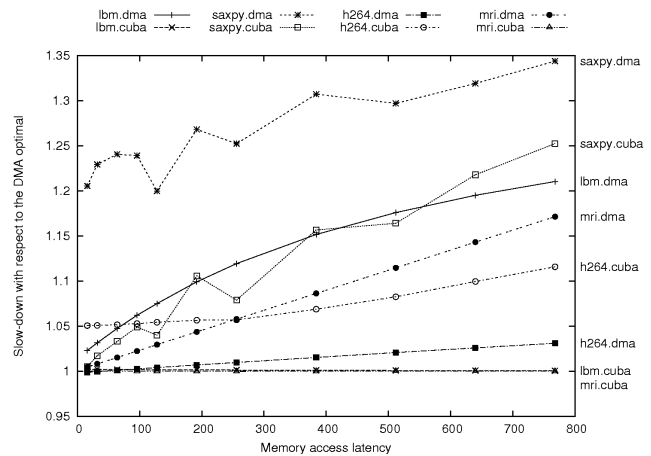


**Figure 7: Number of accesses to the CPU main memory or the co-processor local storage per access to the L2 cache. L2 write-backs and main memory reads are accesses to the CPU main memory. L2 write-through and local memory reads are accesses to the co-processor local storage. DMA read/write operations involve accessing to the CPU main memory and the co-processor local storage**

co-processor. Since CUBA implements a write-through policy for the co-processor data, in CUBA these L2 write-backs do not occur. Moreover, the write-no-allocate policy for the co-processor data increases the number of cache lines in the L2 that can be used to cache other data because many co-processor data are only written (but not read) by the processor. As a result, the number of L2 misses in CUBA is smaller than in the DMA and DMA optimal configurations. These are the main sources of the slight speed-up provided by CUBA with respect to the optimal DMA configuration.

The performance of CUBA for *mri* is near the DMA optimal configuration. In CUBA, the co-processor input set is transferred directly from the disk to the co-processor memory. CUBA allows DMA transfers from I/O devices to the co-processor local memory because the co-processor local memory is mapped in the system physical address space. The *mri* code requires two transfers when using DMA. First the code reads the data from the disk to main memory. A second transfer copies the data from main memory to the co-processor memory. Finally the code calls the co-processor. Figure 7 shows that DMA requires 190 times more memory accesses than CUBA due to this two-step I/O transfer process to the co-processor local storage.

In *saxpy*, CUBA produces the largest performance improvement. In this benchmark the co-processor is called thousands of times (see Table 2). The data transfer time is a significant portion (17.16%) of the total execution time in the DMA version where there are thousands of DMA transfers. Each DMA transfer requires a polling loop, which might take few cycles to detect the end of each transfer. These extra cycles add to the DMA transfer time. In CUBA more data is transferred to the co-processor local memory in the form of L2 write-through actions. Data transfers in CUBA are overlapped with CPU computation before the co-processor is invoked and the transfers do not require polling.



**Figure 8: Slow-down with respect to the dma-opt configuration for different memory latencies**

## 5.2 Memory Latency

Figure 8 shows the slow-down with respect to the *dma-opt* configuration for different memory access latencies. Many existing co-processors, such as GPUs, include GDDR memories which deliver higher bandwidth compared to regular DDR memories. However, we conservatively assume the same latency for both the CPU main memory and the co-processor local storage.

The co-processor execution time depends on the memory latency. We assume a linear dependency between the co-processor execution time and the co-processor local storage latency. We use this approximation because we only have actual kernel speedups for estimating the memory latency of the NVIDIA GeForce 8800 GTX used in [18]. For each kernel we use the memory access to computation ratio reported in [18] as scaling factor for the linear variable.

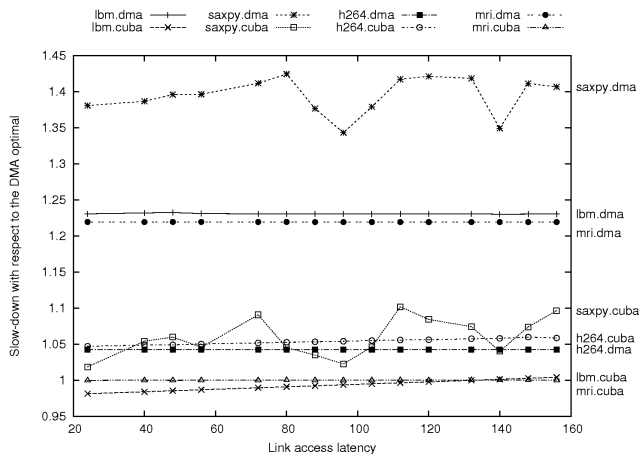
CUBA scales better than DMA for *lbm* and *mri*. Moreover, for all simulated memory latencies, CUBA performs better than DMA in both benchmarks. CUBA scales better for *mri* because there are fewer accesses to the CPU main memory, as shown in Figure 7. In *lbm*, although CUBA requires more accesses to the local storage than DMA, it reduces the number of accesses to main memory (see Figure 7), as discussed previously.

Figure 8 shows that CUBA scales worse than DMA for *saxpy* and *h264*. As show in Figure 7 CUBA increases the number of memory accesses, both to main memory and to the co-processor local. For the longest simulated memory latency (786 cycles), CUBA still performs better than DMA in *saxpy* but is 8.21% worse than DMA for *h264* because of the extra write-through traffic.

## 5.3 Link Latency

Figure 9 shows the slow-down with respect to the *dma-opt* configuration as the latencies of the link interconnecting the CPU and the co-processor vary from 20 to 100 cycles.

The results show that the *lbm* performance only varies 1.8% for *cuba*, while it is constant for *dma*. A similar behavior is found in *h264*, but in this case, it varies less than 1%. Figure 7 shows that for these two benchmarks, CUBA requires accessing the co-processor local storage more often than DMA. Hence, the effect of increasing the link latency



**Figure 9: Slow-down with respect to the dma-opt configuration for different link latencies**

is higher in CUBA. For *mri* the performance is constant for all the simulated link latencies, with little contribution of data transfers to the total execution time.

The performance figure for *saxpy* presents a very irregular pattern. The execution time for each invocation of the kernels implemented in the co-processor for *saxpy* is relatively short, but they are called thousands of times during execution (see Table 2). Each kernel call requires several non-cacheable writes to the co-processor memory-mapped registers that use the link. The CPU will then execute the *scop* instruction which prevents the CPU from fetching new instructions until it commits (see Section 3). As a result, the processor does not execute any other instructions until the non-cacheable writes reach the co-processor. For *h264*, *lbm*, and *mri* the time spend waiting for these memory instructions to complete is less than 0.01%. However, in *saxpy* the CPU is stalled about 50% of the total execution time. Hence, the variability in the stall time is a very important part of the total execution time.

Notice that the slow-down with respect to the *dma-opt* configuration in the benchmark decreases for certain link-latency steps. This effect is mainly due to the non-cacheable accesses to the co-processor registers in the CPU polling loop while waiting for the co-processors. For certain link latency values, the increased time for accessing the co-processor registers results in one more iteration of the polling loop. Because the co-processor is called thousands of times, even the single extra iteration has a visible effect on the total execution time. Because the relative increment to the total execution time for *dma-opt* configuration is higher than *dma* and *cuba*, the ratio plotted in Figure 9 is reduced.

## 6. Related Work

In this section we discuss previous implementations of CPU/co-processor systems and provide examples of co-processors being used to exploit data-level parallelism.

### 6.1 Co-processor Architectures

Examples of fine-grained co-processors that are currently integrated into processors include SIMD and floating-point units, such as the SSE [11] and 3DNow! [1] instructions found in x86 CPUs, and more general interfaces, such as the

MIPS [16] and ARM [19] co-processor interfaces. The communication between integrated co-processors and the CPU is done by means of registers and is controlled by instructions that extend the ISA. There are several proposals for integrating more flexible and coarse-grained co-processors in a similar way. For instance, Chimaera integrates reconfigurable logic as a functional unit inside the core [8] which is able to access the general-purpose registers of the processor and perform arbitrary arithmetic functions. In CUBA, we restrict the focus of co-processors to data-parallel kernel acceleration where the kernel set of each kernel far exceeds the number of registers available and instead choose a model where co-processors have their own local memory for transferring data between the CPU and the co-processor.

There are examples of CPU/co-processor interconnects in the field of reconfigurable co-processors. Garp [9] connects a reconfigurable co-processor to the processor registers and the data cache. MorphoSys [20] allows co-processors to access the system main memory using DMA. OneChip [12] connects a reconfigurable co-processor to the main memory controller instead of to the system bus while providing a mechanism to keep the co-processor local memory and the system memory coherent. CUBA avoids the need to keep coherence by hosting the data in its local memory, ensuring that all input data is available to the co-processor before computation begins. Co-processors in CUBA do not access main memory, but only their own local storage.

As an alternative to tightly integrated co-processors, there are architectures where co-processors are external to the processor, whose memory and registers are mapped into the processor address space. Co-processors can be implemented on-chip, similar to the SPEs in the Cell BE processor [5] or reconfigurable co-processors in the Virtex-II Pro from Xilinx [21]. On-chip co-processors are typically attached to the same interconnection network as the memory and I/O controllers. The NVIDIA GeForce 8 Series is an off-chip architecture where the co-processor is attached to a Peripheral Component Interconnect Express (PCIe) bus. The Cray XD1 connects an FPGA chip to the processor using a direct connection through a dedicated I/O bus [4].

Unlike related architectures, CUBA does not constrain the co-processor memory to be mapped into a given range of the physical address space. In CUBA, the co-processor local memory is dynamically mapped to whatever address space in the process virtual address space required by the application. When a given mapping is set in CUBA, the co-processor local memory actually hosts the same data that system memory in that range would.

### 6.2 Data Transfer

The two preliminary steps for mapping applications into a data-parallel architecture are profiling to identify the computation intensive parts of the applications [3] and a detailed analysis of these parts to determine, for a given co-processor architecture, if they are suitable to be implemented by a co-processor [7]. With these analyses in hand, the application is modified to perform data transfers and synchronization between the CPU and the co-processor. There are frameworks that help automate these tasks such as CIGAR [13].

Coarse-grained co-processor architectures implement data transfer as a marshalling process and a copy of the input data from the main memory to the co-processor local memory and vice versa using DMA [14]. For instance, Mor-

phoSys [20], the Cell processor [5], and the NVIDIA GeForce 8 Series include DMA engines to copy data between the co-processor and the main memory. CUBA avoids data marshalling by hosting the data structures used by co-processors in their local memory. Input data is written directly to the co-processor memory while it is being produced by the CPU.

Garp [9] and OneChip [12] avoid data copying by allowing the co-processor to access the CPU memory hierarchy. This approach requires implementing a memory controller for each co-processor that must implement the memory coherence protocol, perform memory translation, and ensure protection. In CUBA, the OS makes use of the MMU to ensure that the co-processor cannot affect any data other than the appropriate virtual locations of the process currently mapped to its co-processor memory.

Guo [7] identified the data transfer process as a bottleneck in co-processor architectures. He proposes the *smart buffer*, a compiler technique that minimizes the data to be copied [6]. A smart buffer compiler exploits the fact that the input data on consecutive calls to a given co-processor frequently share items with previous calls; these items do not need to be copied. Similar techniques for propagating values in shared memory multiprocessors, such as data forwarding [15], can be used. CUBA allows data to be hosted by the co-processor local storage and uses a hybrid write-through/write-back L2 cache policy. CUBA produces a similar buffering effect, but it is complementary to these techniques.

## 7. Conclusions and Future Work

Co-processors represent an opportunity to perform the data-parallel portions of applications much faster and more efficiently than CPUs. However, managing data communication between the CPU and co-processors is a challenging problem. In order to cope with data communication overheads, applications use techniques such as double-buffering, which usually implies rewriting the code for the application, increased resource usage, and additional overheads that must be amortized by co-processor speedups.

We presented CUBA, an architecture for CPU/co-processors that allows overlapping data communication and computation while requiring few changes to applications. The CUBA model is based on mapping the data structures required by the co-processor into a memory local to the co-processor. Thus, the CPU effectively accesses the co-processor memory whenever it must read or modify shared data structures. The co-processor memory is not accessed directly by the CPU. Instead, the co-processor memory is accessed through the cache hierarchy, so repeated accesses by the CPU to the same data are not penalized.

We evaluated CUBA by comparing it to existing DMA-based architectures using data-parallel benchmarks. In most cases, CUBA performs better than DMA-based architectures while requiring minimal changes to the code.

There is room for further optimization in CUBA. Applications using CUBA must implement a polling loop to wait for the co-processor. Furthermore, CUBA may slow down the portion of the application that is executed on the CPU. Our future work will focus on new mechanisms that allow for implicit synchronization that avoids polling loops. We will add support for fine grained mappings and discuss the support required for multi-core architectures.

## Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN-2007-60625, the Framework Programme 6 HiPEAC NoE (IST-004408). The authors acknowledge the support of the Gigascale Systems Research Center, one of the five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. The authors would also like to thank Mateo Valero, Yale N. Patt, and Alex Ramirez for their insightful comments.

## References

- [1] AMD Staff. *AMD64 Architecture Programmer's Manual*. AMD Corporation, Sept. 2006.
- [2] D. Anderson. *HyperTransport System Architecture*. Addison-Wesley Professional, 2003.
- [3] R. Enzler, M. Platzter, C. Plessl, L. Thiele, and G. Troester. Reconfigurable processors for handhelds and wearables: Application analysis. In *Reconfigurable Technology*, pages 135–146, Denver, CO, USA, Aug. 2001.
- [4] M. Fahey, S. Alam, T. Dunigan Jr, J. Vetter, and P. Worley. Early Evaluation of the Cray XD1. *Cray User Group Conference*, 2005.
- [5] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.
- [6] Z. Guo, B. Buyukkurt, and W. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. *ACM SIGPLAN Notices*, 39(7):249–256, 2004.
- [7] Z. Guo, W. Najjar, F. Vahid, and K. Vissers. A quantitative analysis of the speedup factors of FPGAs over processors. In *FPGA*, pages 162–170, New York, NY, USA, 2004. ACM Press.
- [8] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimera reconfigurable functional unit. In *FCCM*, pages 87–96. IEEE Computer Society Press, 1997.
- [9] J. R. Hauser and J. Wawryznek. Garp: A MIPS processor with a reconfigurable coprocessor. In *FCCM*, pages 296–299, 1997.
- [10] M. Hummel, M. Krause, and D. O'Flaherty. AMD and HP: Protocol enhancements for tightly coupled accelerators. Press Release, 2007.
- [11] Intel Staff. *Intel 64 and IA-32 Architectures Software Developer's Manuals*. Intel, May 2007.
- [12] J. A. Jacob and P. Chow. Memory interfacing and instruction specification for reconfigurable processors. In *FPGA*, pages 145–154, New York, NY, USA, 1999.
- [13] J. H. Kelm, I. Gelado, M. J. Murphy, N. Navarro, S. Lumetta, and W. mei W. Hwu. CIGAR: Application partitioning for a cpu/coprocessor architecture. In *PACT*, pages 317–326, New York, NY, USA, 2007. ACM Press.
- [14] D. Kim, R. Managuli, and Y. Kim. Data cache and direct memory access in programming mediaprocessors. *IEEE Micro*, 21(4):33–42, 2001.
- [15] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. In *ICS*, pages 255–264, New York, NY, USA, 1995. ACM Press.
- [16] MIPS Staff. *MIPS32 Architecture for Programmers*. MIPS Technologies, Mar. 2001.
- [17] J. Renau, B. Fragela, J. Tuck, W. Liu, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator. <http://sesc.sourceforge.net>, Jan. 2005.
- [18] S. Ryoo, C. Rodrigues, S. S. Baghsorhki, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, pages 73–82, 2008.
- [19] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [20] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.
- [21] Xilinx. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, Oct. 2005.