An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems

Isaac Gelado Javier Cabezas
Nacho Navarro
Universitat Politecnica de Catalunya
{igelado, jcabezas, nacho}@ac.upc.edu

John E. Stone Sanjay Patel
Wen-mei W. Hwu
University of Illinois
{jestone, sjp, hwu}@illinois.edu

Abstract

Heterogeneous computing combines general purpose CPUs with accelerators to efficiently execute both sequential control-intensive and data-parallel phases of applications. Existing programming models for heterogeneous computing rely on programmers to explicitly manage data transfers between the CPU system memory and accelerator memory.

This paper presents a new programming model for heterogeneous computing, called *Asymmetric Distributed Shared Memory* (ADSM), that maintains a shared logical memory space for CPUs to access objects in the accelerator physical memory but not vice versa. The asymmetry allows light-weight implementations that avoid common pitfalls of symmetrical distributed shared memory systems. ADSM allows programmers to assign data objects to performance critical methods. When a method is selected for accelerator execution, its associated data objects are allocated within the shared logical memory space, which is hosted in the accelerator physical memory and transparently accessible by the methods executed on CPUs.

We argue that ADSM reduces programming efforts for heterogeneous computing systems and enhances application portability. We present a software implementation of ADSM, called GMAC, on top of CUDA in a GNU/Linux environment. We show that applications written in ADSM and running on top of GMAC achieve performance comparable to their counterparts using programmer-managed data transfers. This paper presents the GMAC system and evaluates different design choices. We further suggest additional architectural support that will likely allow GMAC to achieve higher application performance than the current CUDA model.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management—Distributed Memories

General Terms Design, Experimentation, Performance

Keywords Heterogeneous Systems, Data-centric Programming Models, Asymmetric Distributed Shared Memory

1. Introduction

Maximizing multi-thread throughput and minimizing single-thread latency are two design goals that impose very different and often conflicting requirements on processor design. For example, the Intel Xeon E7450 [28] processor consists of six processor cores each of which is an high-frequency out-of-order, multi-instruction-issue processor with a sophisticated branch prediction mechanism to achieve short single-thread execution latency. This is in contrast to the NVIDIA Tesla GT200 Graphics Processing Unit (GPU) [35] design that achieves high multi-thread throughput with many cores, each of which is a moderate-frequency, multi-threaded, in-order processor that shares its control unit and instruction cache with seven other cores. For control intensive code, the E7450 design can easily outperform the NVIDIA Tesla. For massively data parallel applications, the NVIDIA Tesla design can easily achieve higher performance than the E7450.

Data parallel code has the property that multiple instances of the code can be executed concurrently on different data. Data parallelism exists in many applications such as physics simulation, weather prediction, financial analysis, medical imaging, and media processing. Most of these applications also have control-intensive phases that are often interleaved between data-parallel phases. Hence, general purpose CPUs and accelerators can be combined to form heterogeneous parallel computing systems that efficiently execute all application phases [39]. There are many examples of successful heterogeneous systems. For example, the RoadRunner supercomputer couples AMD Opteron processors with IBM PowerXCell accelerators. If the RoadRunner supercomputer were benchmarked using only its general purpose CPUs, rather than being the top-ranked system in the Top500 List (June 2008), it would drop to a 50th-fastest ranking [6].

Current commercial programming models of processor-accelerator data transfer are based on Direct Memory Access (DMA) hardware, which is typically exposed to applications programmers through memory copy routines. For example, in the CUDA programming model [38], an application programmer can transfer data from the processor to the accelerator device by calling a memory copy routine whose input parameter includes a source data pointer to the processor memory space, a destination pointer to the accelerator memory space, and the number of bytes to be copied. The memory copy interface ensures that the accelerator can only access the part of the application data that is explicitly requested by the memory copy parameters.

In this paper we argue that heterogeneous systems benefit from a data-centric programming model where programmers assign data objects to performance critical methods. This model provides the run-time system with enough information to automatically transfer data between general purpose CPUs and accelerators. Such a

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-839-1/10/03…$10.00

run-time system improves programmability and compatibility of heterogeneous systems. This paper introduces the *Asymmetric Distributed Shared Memory* (ADSM) model, a data-centric programming model, that maintains a shared logical memory space for CPUs to access objects in the accelerator physical memory but not vice versa. This asymmetry allows all coherence and consistency actions to be executed on the CPU, allowing the use of simple accelerators. This paper also presents GMAC, a user-level ADSM library, and discusses design and implementation details of such a system. Experimental results using GMAC show that an ADSM system makes heterogeneous systems easier to program without introducing performance penalties.

The main contributions of this paper are: (1) the introduction of ADSM as a data-centric programming model for heterogeneous systems. The benefits of this model are architecture independence, legacy support, and efficient I/O support; (2) a detailed discussion about the design of an ADSM system, which includes the definition of the necessary API calls and the description of memory coherence and consistency required by an ADSM system; (3) a description of the software techniques required to build an ADSM system for current accelerators on top of existing operating systems; (4) an analysis of different coherence protocols that can be implemented in an ADSM system.

This paper is organized as follows. Section 2 presents the necessary background and motivates this work. ADSM is presented as a data-centric programming model in Section 3, which also discusses the benefit of ADSM for heterogeneous systems and presents the API, consistency model, and different coherence protocols for ADSM systems. The design and implementation of an ADSM system, GMAC, is presented in Section 4. Section 5 presents experimental results. ADSM is compared to other work in Section 6. Finally, Section 7 concludes this paper.

2. Background and Motivation

2.1 Background

General purpose CPUs and accelerators can be coupled in many different ways. Fine-grained accelerators are usually attached as functional units inside the processor pipeline [21, 22, 41, 43]. The Xilinx Virtex 5 FXT FPGAs include a PowerPC 440 connected to reconfigurable logic by a crossbar [46]. In the Cell BE chip, the Synergistic Processing Units, L2 cache controller, the memory interface controller, and the bus interface controller are connected through an Element Interconnect Bus [30]. The Intel Graphics Media Accelerator is integrated inside the Graphics and Memory Controller Hub that manages the flow of information between the processor, the system memory interface, the graphics interface, and the I/O controller [27]. AMD Fusion chips will integrate CPU, memory controller, GPU, and PCIe Controller into a single chip. A common characteristic among Virtex 5, Cell BE, Graphics Media Accelerator, and AMD Fusion is that general purpose CPUs and accelerators share access to system memory. In these systems, the system memory controller deals with memory requests coming from both general purpose CPUs and accelerators.

Accelerators and general purpose CPUs impose very different requirements on the system memory controller. General purpose CPUs are designed to minimize the instruction latency and typically implement some form of strong memory consistency (e.g., sequential consistency in MIPS processors). Accelerators are designed to maximize data throughput and implement weak forms of memory consistency (e.g. Rigel implements weak consistency [32]). Memory controllers for general purpose CPUs tend to implement narrow memory buses (e.g. 192 bits for the Intel Core i7) compared to data parallel accelerators (e.g. 512 bits for the NVIDIA GTX280) to minimize the memory access time. Relaxed consistency models implemented by accelerators allow memory controllers to serve several requests in a single memory access. Strong consistency models required by general purpose CPUs do not offer the same freedom to rearrange accesses to system memory. Memory access scheduling in the memory controller has different requirements for general purpose CPUs and accelerators (i.e., latency vs throughput). Virtual memory management also tends to be quite different on CPUs and accelerators (e.g., GPUs tend to benefit more from large page sizes than CPUs), which makes the design of TLBs and MMUs quite different (e.g., incompatible memory page sizes). Hence, general purpose CPUs and accelerators are connected to separate memories in most heterogeneous systems, as shown in Figure 1. Many such examples of heterogeneous systems currently exist. The NVIDIA GeForce graphics card [35] includes its own GDDR memory (up to 4GB) and is attached to the CPU through a PCIe bus. Future graphics cards based on the Intel Larrabee [40] chip will have a similar configuration. The Roadrunner supercomputer is composed of nodes that include two AMD Opteron CPUs (IBM BladeCenter LS21) and four PowerXCell chips (2x IBM BladeCenter QS22). Each LS21 BladeCenter is connected to two QS22 BladeCenters through a PCIe bus, constraining processors to access only on-board memory [6]. In this paper we assume a base heterogeneous system in Figure 1. However, the concepts developed in this paper are equally applicable to systems where general purpose CPUs and accelerators share the same physical memory.

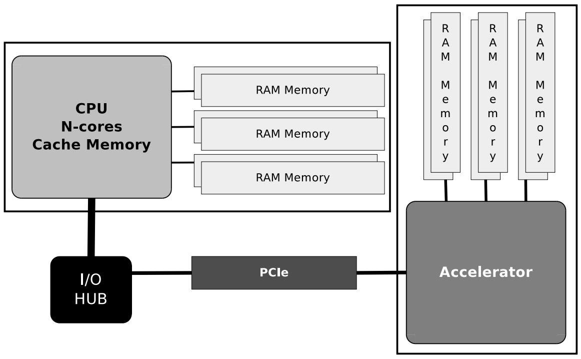


Figure 1. Reference Architecture, similar to desktop GPUs and RoadRunner blades

2.2 Motivation

Heterogeneous parallel computing improves application performance by executing computationally intensive data-parallel kernels on accelerators designed to maximize data throughput, while executing the control-intensive code on general purpose CPUs. Hence, some data structures are likely to be accessed primarily by the code executed by accelerators. For instance, execution traces show that about 99% of read and write accesses to the main data structures in the *NASA Parallel Benchmarks* (NPB) occur inside computationally intensive kernels that are amenable for parallelization.

Figure 2 shows our estimation for the average memory bandwidth requirements for the computationally intensive kernels of some NPB benchmarks, assuming a 800MHz clock frequency for different values of IPC and illustrates the need to store the data structures required by accelerators in their own memories. For instance, if all data accesses are done through a PCIe bus[1], the maximum achievable value of IPC is 50 for `bt` and 5 for `ua`, which

[1] If both accelerator and CPU share the same memory controller, the available accelerator bandwidth will be similar to HyperTransport in Figure 2, which also limits the maximum achievable value of IPC.

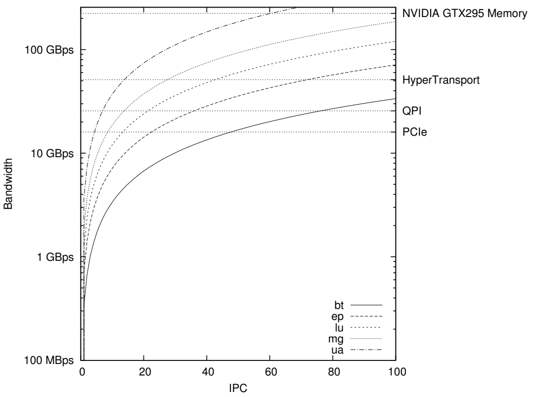


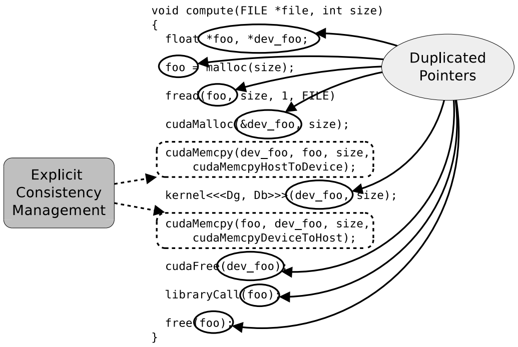
Figure 2. Estimated bandwidth requirements for computationally intensive kernels of `bt`, `ep`, `lu`, `mg`, `ua` benchmarks, assuming a 800MHz clock frequency

is a very small fraction of the peak execution rate of the NVIDIA GTX295 GPU. In all cases, the level of IPC that can be supported by the GTX295 memory bandwidth is much higher than the supported by PCIe or similar interconnect schemes. In order to achieve optimal system performance, it is crucial to host data structures accessed by computationally intensive kernels in on-board accelerator memories.

These results discourage the implementation of fully-coherent heterogeneous system due to the high number of coherence requests produced by accelerators during kernel execution (e.g., invalidation requests the first time data initialized by the CPU is accessed by accelerators). Moreover, a fully coherent heterogeneous system would require both, CPUs and accelerators to implement the very same coherence protocol. Hence, it would be difficult, if not infeasible, to use the same accelerator (e.g., a GPU) in systems based on different CPU architectures, which would impose a significant economic penalty on accelerator manufactures. Finally, the logic required to implement the coherence protocol in the accelerator would consume a large silicon area, currently devoted to processing units, which would decrease the benefit of using accelerators.

The capacity of on-board accelerator memories is growing and currently allows for many data structures to be hosted by accelerators memories. Current GPUs use 32-bit physical memory addresses and include up to 4GB of memory and soon GPU architectures will move to larger physical addresses (e.g., 40-bit in NVIDIA Fermi) to support larger memory capacities. IBM QS20 and QS21, the first systems based on Cell BE, included 512MB and 1GB of main memory per chip respectively. IBM QS22, the latest Cell-based system, supports up to 16GB of main memory per chip. IBM QS22 is based on the PowerXCell 8i chip, which is an evolution of the original Cell BE chip, modified to support a larger main memory capacity [6]. These two examples illustrate the current trend that allows increasingly larger data structures to be hosted by accelerators and justifies our ADSM design.

Programming models for current heterogeneous parallel systems, such as NVIDIA CUDA [11] and OpenCL [1], present different memories in the system as distinct memory spaces to the programmer. Applications explicitly request memory from a given memory space (i.e. `cudaMalloc()`) and perform data transfers between different memory spaces (i.e. `cudaMemcpy()`). The example in Figure 3 illustrates this situation. First, system memory is allocated (`malloc()`) and initialized (`fread()`). Then, accelerator memory is allocated (`cudaMalloc()`) and the data structure is copied to the accelerator memory (`cudaMemcpy()`), before code is executed on the accelerator. OpenCL and Roadrunner codes do not significantly differ from code in Figure 3.



```
void compute(FILE *file, int size)
{
  float *foo, *dev_foo;
  foo = malloc(size);
  fread(foo, size, 1, FILE)
  cudaMalloc(&dev_foo, size);
  cudaMemcpy(dev_foo, foo, size,
      cudaMemcpyHostToDevice);
  kernel<<<Dg, Db>>>(dev_foo, size);
  cudaMemcpy(foo, dev_foo, size,
      cudaMemcpyDeviceToHost);
  cudaFree(dev_foo);
  libraryCall(foo);
  free(foo);
}
```

Figure 3. Example code with duplicated pointers and explicit consistency management

Such programming models ensure that data structures reside in the memory of the processor (CPU or accelerator), that performs subsequent computations. These models also imply that programmers must explicitly request memory on different processors and, thus, a data structure (`foo` in Figure 3) is referenced by two different memory addresses: `foo`, a virtual address in system memory, and `dev_foo`, a physical address in the accelerator memory. Programmers must explicitly manage memory coherence (e.g., with a call to `cudaMemcpy()`) before executing kernels on the accelerator. This approach also prevents parameters from being passed by reference to accelerator kernels [19] and computationally critical methods to return pointers to the output data structures instead of returning the whole output data structure, which would save bandwidth whenever the code at CPU only requires accessing a small portion of the returned data structure. These approaches harm portability because they expose data transfer details of the underlaying hardware. Offering a programming interface that requires a single allocation call and removes the need for explicit data transfers would increase programmability and portability of heterogeneous systems and is the first motivation of this paper.

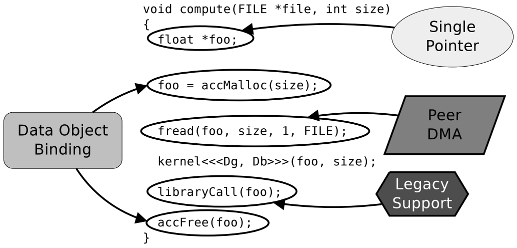
The cost of data transfers between general purpose CPUs and accelerators might eliminate the benefit of using accelerators. Double buffering can help to alleviate this situation by transferring parts of the data structure while other parts are still in use. In the example of Figure 3, the input data would be read iteratively using a call to `fread()` followed by an asynchronous DMA transfer to the accelerator memory. Synchronization code is necessary to prevent overwriting system memory that is still in use by an ongoing DMA transfer [19]. The coding effort to reduce the cost of data transfer harms programmability of heterogeneous systems. Automatically overlapping data transfers and CPU computation without code modifications is the second motivation of this paper.

3. Asymmetric Distributed Shared Memory

Asymmetric Distributed Shared Memory (ADSM) maintains a shared logical memory space for CPUs to access objects in the accelerator physical memory but not vice versa. This section presents ADSM as a data-centric programming model and the benefit of an asymmetric shared address space.

3.1 ADSM as a Data-Centric Programming Model

In a data-centric programming model, programmers allocate or declare data objects that are processed by methods, and annotate performance critical methods (*kernels*) that are executed by accelerators. When such methods are assigned to an accelerator, their cor-



```
void compute(FILE *file, int size)
{
  float *foo;
  foo = accMalloc(size);
  fread(foo, size, 1, FILE);
  kernel<<<Dg, Db>>>(foo, size);
  libraryCall(foo);
  accFree(foo);
}
```

Figure 4. Usage example of ADSM

responding data objects are migrated to on-board accelerator memory. Figure 4 shows an example using such a model, based on the code from Figure 3, and illustrates the main benefits of a data-centric programming model with ADSM for heterogeneous systems: architecture-independence, legacy-code support, and peer DMA.

ADSM removes the need to explicitly request memory on different memory spaces. Programmers assign data objects to methods that might or might not be executed by accelerators. Run-time systems can be easily built under this programming model to automatically assign methods and their data objects to accelerators, if they are present in the system. High performance systems are likely to continue having separate physical memories and access paths for CPUs and accelerators. However, CPUs and accelerators are likely to share the same physical memory and access path in low-cost systems. An application written following a data-centric programming model will target both kinds of systems efficiently. When the application is run on a high performance system, accelerator memory is allocated and the run-time system transfers data between system memory and accelerator memory when necessary. In the low-cost case, system memory (shared by CPU and accelerator) is allocated and no transfer is done. Independence from the underlying hardware is the first benefit provided by ADSM.

ADSM offers a convenient software migration path for existing applications. Performance critical libraries and application code are moved to accelerator engines, leaving less critical code porting for later stages [6]. CPUs and accelerators do not need to be ISA-compatible to interoperate, as long as data format and calling conventions are consistent. This programming model provides the run-time system with the information necessary to make data structures accessible to the code that remains for execution by the CPU. This is the second gain offered by a data-centric programming model.

Data objects used by kernels are often read from and written to I/O devices (e.g., a disk or network interface). ADSM enables data structures used by accelerators to be passed as parameters to the corresponding system calls that read or write data for I/O devices (e.g. `read()` or `write()`). If supported by the hardware, the run-time system performs DMA transfers directly to and from accelerator memory (*peer DMA*), otherwise an intermediate buffer in system memory might be used. Applications benefit from *peer DMA* without any source code modifications, which is the third advantage of this programming model.

3.2 ADSM Run-time Design Rationale

As discussed in Sec. 2.2, distributed memories are necessary to extract all the computational power from heterogeneous systems. However, a data-centric programming model hides the complexity of this distributed memory architecture from programmers.

A DSM run-time system reconciles physically distributed memory and logical shared memory. Traditional DSM systems are prone to thrashing, which is a performance limiting factor. Thrashing in DSM systems typically occurs when two nodes compete for write access to a single data item, causing data to be transferred back and forth at such a high rate that no work can be done. Access to synchronization variables, such as locks and semaphores, tends to be a primary cause of thrashing in DSM. This effect is unlikely to occur in heterogeneous systems because hardware interrupts are typically used for synchronization between accelerators and CPUs and, therefore, there are no shared synchronization variables.

API Call	Description
adsmAlloc(size)	Allocates *size* bytes of shared memory and returns the shared memory address where the allocated memory begins.
adsmFree(addr)	Releases a shared memory region that was previously allocated using *adsmAlloc()*.
adsmCall(kernel)	Launches the execution of method *kernel* in an accelerator.
adsmSync()	Yields the CPU to other processes until a previous accelerator calls finishes.

Table 1. Compulsory API calls implemented by an ADSM run-time

If synchronization variables are used (e.g., polling mode), special hardware mechanisms are typically used. These special synchronization variables are outside the scope of our ADSM design. False sharing, another source of thrashing in DSM, is not likely to occur either because sharing is done at the data object granularity.

A major issue in DSM is the memory coherence and consistency model. DSM typically implements a relaxed memory consistency model to minimize coherence traffic. Relaxed consistency models (e.g., release consistency) reduce coherence traffic at the cost of requiring programmers to explicitly use synchronization primitives (e.g. *acquire* and *release*). In a data-centric programming model memory consistency is only relevant from the CPU perspective because all consistency actions are driven by the CPU at method call and return boundaries. Shared data structures are released by the CPU when methods using them are invoked, and data items are acquired by the CPU when methods using them return.

DSM pays a performance penalty for detection of memory accesses to shared locations that are marked as invalid or dirty. Memory pages that contain invalid and dirty items are marked as not present in order to get a *page fault* whenever the program accesses any of these items. This penalty is especially important for faults produced by performance critical code whose execution is distributed among the nodes in the cluster. Data cannot be eagerly transferred to each node to avoid this performance penalty because there is no information about which part of the data each node will access. This limitation is not present in ADSM since the accelerator executing a method will access only shared data objects explicitly assigned to the method.

The lack of synchronization variables hosted by shared data structures, the implicit consistency primitives at call/return boundaries, and the knowledge of data structures accessed by performance critical methods are the three reasons that lead us to design an ADSM as an efficient way to support a data-centric programming model on heterogeneous parallel systems.

3.3 Application Programming Interface and Consistency Model

We identify four fundamental functions an ADSM system must implement: shared-data allocation, shared-data release, method invocation, and return synchronization. Table 1 summarizes these necessary API calls.

Shared-data allocation and release calls are used by programmers to declare data objects that will be used by kernels. In its simplest form, the allocation call only requires the size of the data structure and the release requires the starting memory address for the data structure to be released. This minimal implementation assumes that any data structure allocated through calls to the ADSM

API will be used by all accelerator kernels. A more elaborate scheme would require programmers to pass one or more method identifiers to effectively assign the allocated/released data object to one or more accelerator kernels.

Method invocation and return synchronization are already found in many heterogeneous programming APIs, such as CUDA. The former triggers the execution of a given kernel in an accelerator, while the latter yields the CPU until the execution on the accelerator is complete.

ADSM employs a *release consistency* model where shared data objects are released by the CPU on accelerator invocation (*adsmCall()*) and acquired by the CPU on accelerator return (*adsmSync()*). This semantic ensures that accelerators always have access to the objects hosted in their physical memory by the time a kernel operates on them. Implicit acquire/release semantics increase programmability because they require fewer source code lines to be written and it is quite natural in programming environments such as CUDA, where programmers currently implement this consistency model manually, through calls to `cudaMemcpy()`.

4. Design and Implementation

This section describes the design and implementation of *Global Memory for ACcelerators* (GMAC), a user-level ADSM run-time system. The design of GMAC is general enough to be applicable to a broad range of heterogeneous systems, such as NVIDIA GPUs or the IBM PowerXCell 8i. We implement GMAC for GNU/Linux based systems that include CUDA-capable NVIDIA GPUs. The implementation techniques presented here are specific to our target platform but they can be easily ported to different operating systems (e.g. Microsoft Windows) and accelerator interfaces (e.g. OpenCL).

ADSM might be implemented using a hybrid approach, where low-level functionalities are implemented in the operating system kernel and high-level API calls are implemented in a user-level library. Implementing low-level layers within the operating system kernel code allows I/O operations involving shared data structures to be fully supported without performance penalties (see Section 4.4). We implement all GMAC code in a user-level library because there is currently no operating system kernel-level API for interacting with the proprietary NVIDIA driver required by CUDA.

By using a software implementation of ADSM, GMAC allows multiple cores in a cache-coherent multi-core CPU to all maintain and access share memories with accelerators.

4.1 Overall Design

Figure 5 shows the overall design of the GMAC library. The lower-level layers (OS Abstraction Layer and Accelerator Abstraction Layer) are operating system and accelerator dependent, and they offer an interface to upper-level layers for allocation of system and accelerator memory, setting of memory page permission bits, transfer of data between system and accelerator memory, invocation of kernels, and waiting for completion of accelerator execution, I/O functions, and data type conversion, if necessary. In our reference implementation the OS Abstraction Layer interacts with POSIX-compatible operating systems such as GNU/Linux. We implement two different Accelerator Abstraction Layers to interact with CUDA-capable accelerators: the CUDA Run-Time Layer and the CUDA Driver Layer. The former interacts with the CUDA runtime, which offers us limited control over accelerators attached to the system. The latter interacts with the CUDA driver, which offers a low-level API and, thus, allows having full control over accelerators at the cost of more complex programming of the GMAC code base. At application load-time, the user can select which of the two different Accelerator Abstraction Layers to use. In Section 5, we use the CUDA Run-Time Layer to compare the performance of GMAC with respect to CUDA and the CUDA Driver Layer to extract a detailed execution time break-down of applications. The top-level layer implements the GMAC API to be used by applications and libraries.

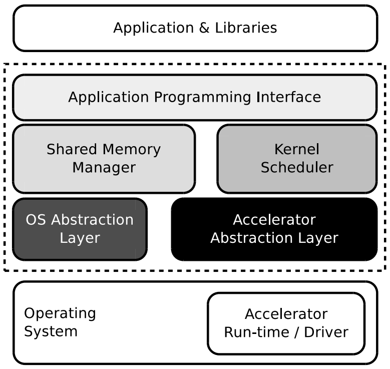


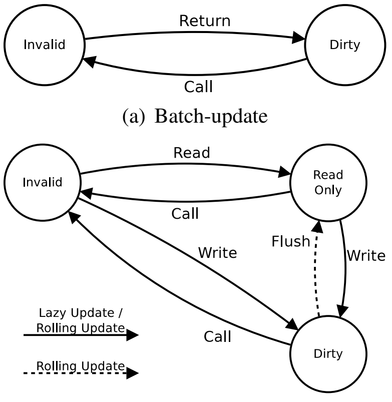
Figure 5. Software layers that conforms the GMAC library.

The Shared Memory Manager in Figure 5 manages shared memory areas and creates the shared address space between host CPUs and accelerators. An independent memory management module allows testing of different management policies and coherence protocols with minor modifications to the GMAC code base. The kernel scheduler selects the most appropriate accelerator for execution of a given kernel, and implements different scheduling policies depending on the execution environment. A detailed analysis of kernel scheduling is out of the scope of the present paper and we refer the reader to Jimenez et al. [29].

4.2 Shared Address Space

GMAC builds a shared address space between the CPUs and the accelerator. When the application requests shared memory (*adsmAlloc()*), accelerator memory is allocated on the accelerator, returning a memory address (virtual or physical, depending on the accelerator) that can be only used by the accelerator. Then, we request the operating system to allocate system memory over the same range of virtual memory addresses. In a POSIX-compliant operating system this is done through the `mmap` system call, which accepts a virtual address as a parameter and maps it to an allocated range of system memory (*anonymous memory mapping*). At this point, two identical memory address ranges have been allocated, one in the accelerator memory and the other one in system memory. Hence, a single pointer can be returned to the application to be used by both CPU code and accelerator code.

The operating system memory mapping request might fail if the requested virtual address range is already in use. In our single-GPU target system this is unlikely to happen because the address range typically returned by calls to `cudaMalloc()` is outside the ELF program sections. However, this implementation technique might fail when using other accelerators (e.g. ATI/AMD GPUs) or on multi-GPU systems, where calls to `cudaMalloc()` for different GPUs are likely to return overlapping memory address ranges. A software-based solution for this situation requires two new API calls: *adsmSafeAlloc(size)* and *adsmSafe(address)*. The former allocates a shared memory region, but returns a pointer that is only valid in the CPU code. The latter takes a CPU address and returns the associated address for the target GPU. GMAC maintains the mapping between these associated addresses so that any CPU changes to the shared address region will be reflected in the accelerator memory. Although this software technique works in all cases where shared data structures do not contain embedded pointers, it requires programmers to explicitly call *adsmSafe()* when passing



(a) Batch-update

(b) Lazy-update and Rolling-update

Figure 6. State transition diagram for the memory coherence protocols implemented in GMAC.

references to the accelerator. Since OpenCL does not allow the use of pointers in GPU kernel code, such kernels are already written using relative memory indexing arithmetic.

A good solution to the problem of conflicting address ranges between multiple accelerators is to have virtual memory mechanisms in accelerators. With virtual memory mechanisms in both CPUs and accelerators, the *adsmAlloc()* can be guaranteed to find an available virtual address in both CPU's address space and accelerator's address space. Thus, accelerators and CPUs can always use the same virtual memory address to refer to shared data structures. Additionally, accelerator virtual memory simplifies the allocation of shared data structures. In this case, the implementation of *adsmAlloc()* first allocates system and accelerator memory and fills the necessary memory translation data structures (e.g., a page table) to map the allocated physical system and accelerator memory into the same virtual memory range on the CPU and on the accelerator. Virtual memory mechanisms are implemented in latest GPUs, but not available to programmers [24]

4.3 Memory Coherence Protocols

The layered GMAC architecture allows multiple memory coherence protocols to coexist and enables programmers to select the most appropriate protocol at application load time. The GMAC coherence protocols are defined from the CPU perspective. All bookkeeping and data transfers are managed by the CPU. The accelerators do not perform any memory consistency or coherence actions. This asymmetry allows the use of simple accelerators.

Figure 6 shows the state transition diagrams for the coherence protocols provided by GMAC. In the considered protocols, a given shared memory range can be in one of three different states. *Invalid* means that the memory range is only in accelerator memory and must be transferred back if the CPU reads this memory range after the accelerator kernel returns. *Dirty* means that the CPU has an updated copy of the memory range and this memory range must be transferred back to the accelerator when the accelerator kernel is called. *Read-only* means that the CPU and the accelerator have the same version of the data so the memory region does not need to be transferred before the next method invocation on the accelerator.

Batch-update is a pure write-invalidate protocol. System memory gets invalidated on kernel calls and accelerator memory gets invalidated on kernel return. On a kernel invocation (*adsmCall()*) the CPU invalidates all shared objects, whether or not they are accessed by the accelerator. On method return (*adsmSync()*), all shared objects are transferred from accelerator memory to system memory and marked as dirty, thus implicitly invalidates the accelerator memory. The invalidation prior to method calls requires transferring all objects from system memory to accelerator memory even if they have not been modified by the CPU. The memory manager keeps a list of the starting address and size of allocated shared memory objects in order to perform these transfers. This is a simple protocol that does not require detection of accesses to shared data by the code executed on the CPU. This naive protocol mimics what programmers tend to implement in the early stages of application implementation.

Lazy-update improves upon batch-update by detecting CPU modifications to objects in read-only state and any CPU read or write access to objects in invalid state. These accesses are detected using the CPU hardware memory protection mechanisms (accessible using the `mprotect()` system call) to trigger a page fault exception (delivered as a POSIX signal to user-level), which causes a page fault handler to be executed. The code inside the page fault handler implements the state transition diagram shown in Figure 6(b).

Shared data structures are initialized to a read-only state when they are allocated, so read accesses do not trigger a page fault. If the CPU writes to any part of a read-only data structure, the structure is marked as dirty, and the memory protection state is updated to allow read/write access. Memory protection hardware is configured to trigger a page fault on any access (read or write) to shared data structures in invalid state. Whenever a data structure in invalid state is accessed by the CPU, the object is transferred from accelerator memory to system memory, and the data structure state is updated to read-only, on a read access, or to dirty on a write access.

On a kernel invocation all shared data structures are invalidated and those in the dirty state are transferred from system memory to accelerator memory. On kernel return no data transfer is done and all shared data objects remain in invalid state. This approach presents two benefits: (1) only data objects modified by the CPU are transferred to the accelerator on method calls, and (2) only data structures accessed by the CPU are transferred from accelerator memory to system memory after method return. This approach produces important performance gains with respect to batch-update in applications where the code executed on the accelerator is part of an iterative computation and the code executed on the CPU after the accelerator invocation only updates some of the data structures used or produced by the code executed on the accelerator.

Rolling-update is a hybrid write-update/write-invalidate protocol. Shared data structures are divided into fixed size memory blocks. The memory manager, as in *batch-update* and *lazy-update*, keeps a list of the starting addresses and sizes of allocated shared memory objects. Each element in this list is extended with a list of the starting addresses and sizes of the memory blocks composing the corresponding shared memory object. If the shared object size of any of these blocks is smaller than the default memory block size, the list will include the smaller value. The same memory protection mechanisms that lazy-update uses to detect read and write accesses to dirty and read-only data structures are used here to detect read and write accesses to dirty and read-only blocks. This protocol only allows a fixed number of blocks to be in the dirty state on the CPU, which we refer to as *rolling size*. If the maximum number of dirty blocks is exceeded due to a write access that marks a new block as dirty, the oldest block is asynchronously transferred from system memory to accelerator memory and the block is marked as read-only (dotted line in Figure 6(b)). In our base implementation we use an adaptive approach to set the rolling size: every time a new memory structure is allocated (*adsmAlloc()*), the rolling size is increased by a fixed factor (with a default value of 2 blocks). This approach exploits the fact that applications tend to use all allocated data structures at the same time. Creating a dependence between the number of allocated regions and the maximum number of blocks in the dirty state ensures that, at least, each region might have one of its blocks marked as dirty.

Rolling-update exploits the spatial and temporal locality of accesses to data structures in much the same way that hardware caches do. We expect codes that sequentially initialize accelerator input data structures will benefit from rolling-update. In this case, data is eagerly transfered from system memory to accelerator memory while the CPU code continues producing the remaining accelerator input data. Hence, we expect that rolling-update will automatically overlap data transfers and computation on the CPU. Each time the CPU reads an element that is in invalid state, it fetches only the fixed size block that contains the element accessed. Therefore, rolling update also reduces the amount of data transferred from accelerators when the CPU reads the output kernel data in a scattered way.

All coherence protocols presented in this section contain a potential deficiency. If, after an accelerator kernel returns, the CPU reads a shared object that is not written by the kernel, it must still transfer the data value back from the accelerator memory. Interprocedural pointer analysis [13] in the compiler or programmers can annotate each kernel call with the objects that the kernel will write to, then the objects can remain in read-only or dirty state at accelerator kernel invocation. ADSM enables interprocedural pointer analysis to detect those data structures being accessed by kernels, because both CPU and accelerator use the same memory address to refer to the same memory object.

4.4 I/O and Bulk Memory Operations

An I/O operation might, for instance, read data from disk and write it to a shared object, as in the example in Figure 4 (Section 2.2). First, a page fault occurs because the call to `read()` requires writing to a read-only memory block. Then, GMAC marks the memory block as read/write memory and sets the memory block to the dirty state, so the call to `read()` can proceed. However, when using rolling-update, once the first memory block is read from disk and written to memory, a new page fault exception is triggered because `read()` requires writing to the next memory block of the shared object. However, the second page fault aborts the `read()` function and after handling the second page fault, the `read()` function cannot be restarted because the first block of its data has already been read into the destination. The operating system prevents an ongoing I/O operation from being restarted once data has been read or written. GMAC uses library interposition to overload I/O calls to perform any I/O read and write operations affecting shared data objects in block sized memory chunks and, thus, avoids restarting system calls. GMAC offers the illusion of *peer DMA* to programmers, but the current implementation still requires intermediate copies in system memory.

We use library interposition in GMAC to overload bulk memory operations (i.e. `memset()` and `memcpy()`). The overloaded implementations check if the memory affected by bulk memory operations involve shared objects and they use the accelerator-specific calls (e.g. `cudaMemset()` and `cudaMemcpy()`) for shared data structures, while forwarding calls to the standard C library routines when only system memory is involved. Overloading bulk memory operations avoids unnecessary intermediate copies to system memory and avoids triggering page faults on bulk memory operations.

5. Experimental Results

This Section presents an experimental evaluation of the GMAC library. We run our experiments on a SMP machine with 2 AMD Dual-core Opteron 2222 chips running at 3GHz, and 8GB of RAM memory. A NVIDIA G280 GPU card with 1GB of device memory is attached through a PCIe 2.0 16X bus. The machine runs a Debian GNU/Linux operating system, with Linux kernel 2.6.26, NVIDIA driver 185.18.08, and CUDA 2.2. Application and library code is compiled using the GNU/GCC compiler 4.3 with the -O3 optimization level. Execution times are taken using the `gettimeofday()` system call, which offers a granularity of microseconds.

Benchmark		Description
cp	Coulombic Potential	Computes the coulombic potential at each grid point over on plane in a 3D grid in which point charges have been randomly distributed. Adapted from ‘cionize’ benchmark in VMD.
mri$_{fhd}$	Magnetic Resonance Imaging FHD	Computation of an image-specific matrix F_d^H, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.
mri$_q$	Magnetic Resonance Imaging Q	Computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.
pns	Petri Net Simulation	Implements a generic algorithm for Petri net simulation. Petri nets are commonly used to model distributed systems.
rpes	Rys Polynomial Equation Solver	Calculates 2-electron repulsion integrals which represent the Coulomb interaction between electrons in molecules.
sad	Sum of Absolute Differences	Sum of absolute differences kernel, used in MPEG video encoders. Based on the full-pixel motion estimation algorithm found in the JM reference H.264 video encoder.
tpacf	Two Point Angular Correlation Function	TPACF is an equation used here as a way to measure the probability of finding an astronomical body at a given angular distance from another astronomical body.

Table 2. Parboil Benchmarks Description

We use the Parboil Benchmark Suite [26], described in Table 2, to take performance metrics. Micro-benchmarks are also used to measure the overheads produced by GMAC when Parboil benchmarks are not sufficient. In all experiments, each benchmark is executed 16 times and average values are used. We use the CUDA Run-time Abstraction Layer to compare the performance of GMAC with CUDA. For other experiments we use the CUDA Driver Abstraction Layer to discard CUDA initialization time.

The porting time from CUDA to GMAC for the seven benchmarks included in Parboil took less than eight hours of work. The porting process only involved removing code that performed explicit data transfers and handled double allocation of data structures. The porting process did not involve adding any source code lines to any of the benchmarks. After being ported to GMAC, the total number of lines of code decreased in all benchmarks. This is an indicator that an ADSM-based data-centric programming model increases programmability of heterogeneous parallel systems.

5.1 Coherence Protocols

Figure 7 shows the slow-down for all benchmarks included in the Parboil Benchmark Suite with respect to the default CUDA implementation for GMAC using different coherence protocols. The GMAC implementation using the *batch-update* coherence protocol always performs worse than other versions, producing a slow-down of up to 65.18X in *pns* and 18.64X in *rpes*. GMAC implementations

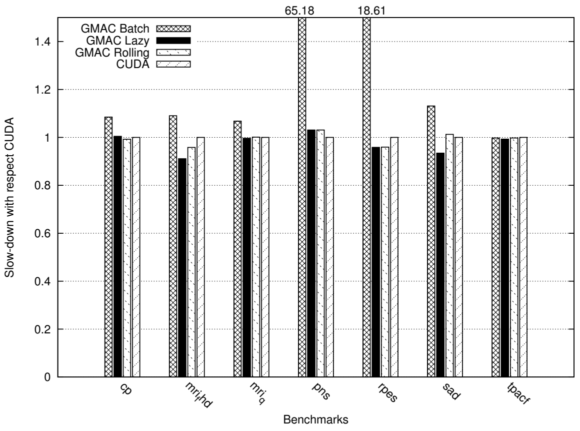


Figure 7. Slow down for different GMAC versions of Parboil benchmarks with respect to CUDA versions

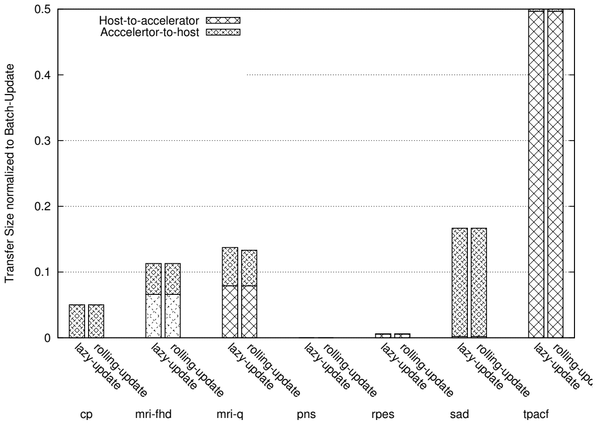


Figure 8. Transferred data by different protocols normalized to data transferred by *Batch-update*

using *lazy-update* and *rolling-update* achieve performance equal to the original CUDA implementation.

Figure 8 shows data transferred by *lazy-update* and *batch-update* normalized to the data transferred by *batch-update*. The *batch-update* coherence protocol produces long execution times because data is transferred back and forth from system memory to accelerator memory on every accelerator invocation. This illustrates the first benefit of a data-centric programming model, where data transfers are automatically handled by the run-time system. Inexperienced programmers tend to take an over-conservative approach at first, transferring data even when it is not needed. A data-centric programming model automates data transfer management and, thus, even initial implementations do not pay the overhead of unnecessary data transfers.

The original CUDA code, *lazy-update*, and *rolling-update* achieve similar execution times. In some benchmarks, there is a small speed-up for GMAC with respect to the CUDA version. This shows the second benefit of GMAC: applications programmed using GMAC perform as well as a hand-tuned code using existing programming models, while requiring less programming effort.

Fine-grained handling of shared objects in *rolling-update* avoids some unnecessary data transfers (i.e. *mri-q* in Figure 8). We use a 3D-Stencil computation to illustrate the potential performance benefits of *rolling-update*. Figure 9 shows the execution time of this benchmark for different input volume sizes and memory block sizes. As we increase the volume size, *rolling-update* offers a greater benefit than *lazy-update*. The 3D-Stencil computation requires introducing a *source* on the target volume on each time-step, which tends to have zero values for most of the volume because it represents a small emitter localized at some point in space. In this version, the CPU executes the code that performs the *source introduction*. *Lazy-update* requires transferring the entire volume prior to introducing the source, while *rolling-update* only requires transferring the few memory blocks that are actually modified by the CPU. This is the main source of performance improvement of *rolling-update* over *lazy-update*.

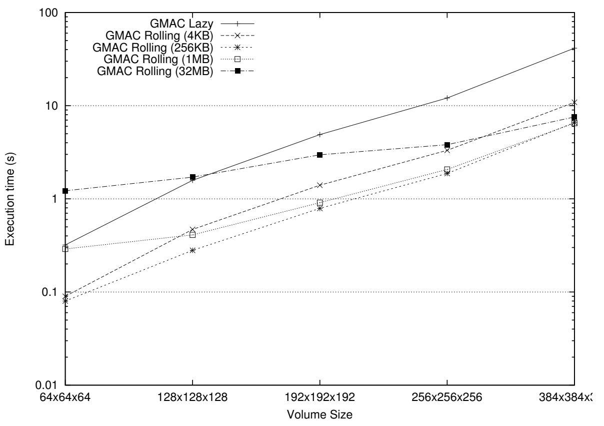


Figure 9. Execution time for a 3D-Stencil computation for different volume sizes

The combining effects of eager data transfer versus efficient bandwidth usage are illustrated in Figure 9 too. This Figure shows that execution times are longer for a memory block size of 32MB than for memory block sizes of 256KB and 1MB, but the difference in performance decreases as the size of the volume increases. Source introduction typically requires only accessing to one single memory block and, hence, the amount of data transferred depends on the memory block size. 3D-Stencil also requires writing to disk the output volume every certain number of iterations and, thus, the complete volume must be transferred from accelerator memory. Writing to disk benefits from large memory block because large data transfers make a more efficient usage of the interconnection network bandwidth than smaller ones. As we increase the volume size, the contribution of the disk write to the total execution time becomes more important and, therefore, a large memory block size reduces the writing time.

These results shows the importance of reducing the amount of transferred data. Figures 7 and 8 show the relationship between the amount of transferred data and the execution time. The largest slow-downs in *batch-update* are produced in those benchmarks that transfer the most data, (*rpes* and *pns*). Programmer annotation and/ or compiler or hardware support to avoid such transfers is a clear need.

Figure 10 shows the break-down of execution time for all benchmarks when using *rolling-update*. Most execution time is spent on computations on the CPU or at the GPU. I/O operations, on those benchmarks that require reading from or writing to disk, and data transfers are the next-most time consuming operations in all benchmarks. The first remarkable fact is that the overhead due to signal handling to detect accesses to non-present and read-only memory blocks is negligible, always below 2% of the total execution time. Figure 10 also shows that some benchmarks (*mri-fhd* and *mri-q*)

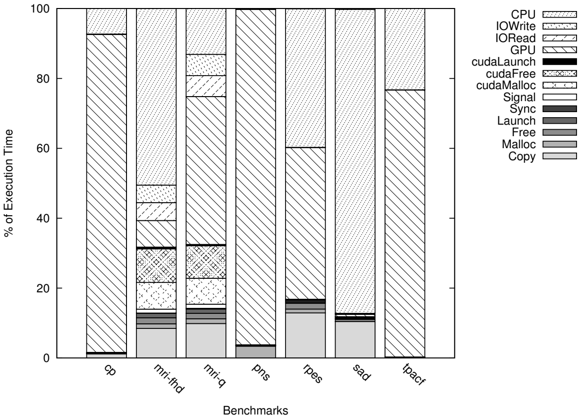


Figure 10. Execution time break-down for Parboil benchmarks

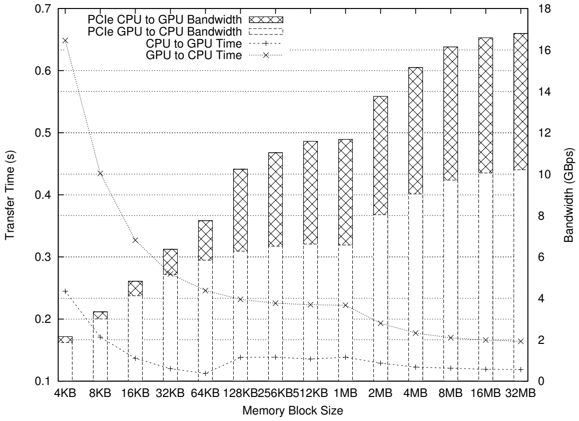


Figure 11. Execution times (lines) and maximum data transfer bandwidth (boxes) for vector addition for different vector and block sizes

have high levels of I/O read activities and would benefit from hardware that supports peer DMA.

5.2 Memory Block Size

The memory block size is a key design parameter of the *rolling-update* protocol. The larger the memory block size, the less page fault exceptions are triggered in the processor. However, a small memory block size is essential to eagerly transfer data from system memory to the accelerator memory and to avoid transferring too much data from accelerator memory to system memory when reading scattered data from the accelerator memory.

Our first experiment consists of running the Parboil benchmarks using different memory block sizes and fixing the maximum number of memory blocks in dirty state. Experimental results show that there is no appreciable difference in the execution time of Parboil benchmarks in this experiment, due to the small contribution by the CPU code accessing accelerator-hosted data to the total execution time.

We use a micro-benchmark that adds up two 8 million elements vectors to show how the execution time varies for different memory block size values. Figure 11 shows the execution time for different block sizes of this synthetic benchmark using different block sizes. We also plot the average transfer bandwidth for each block size. The data transfer bandwidth increases with the block size, reaching its maximum value for block sizes of 32MB. Data transfer times for vector addition decrease as the transfer bandwidth increases. The execution time reduction when moving from memory block sizes from 4KB to 8KB and from 8KB to 16KB is greater than the increase in the data transfer bandwidth. Small memory block sizes produce many page faults to be triggered by the CPU code producing and consuming the data objects. On a page fault, the GMAC code searches for the faulting block in order to modify its permission bits and state. GMAC keeps memory blocks in a balanced binary tree, which requires $O(log_2(n))$ operations to locate a given block. For a fixed data object size, a small memory block size requires more elements to be in the balanced binary tree and, thus, the overhead due to the search time becomes the dominant overhead. A large memory block size allows an optimal utilization of the bandwidth provided by the interconnection network and reduces the overhead due to page faults.

There is an anomaly in Figure 11 for a block size of 64KB. The CPU-to-accelerator transfer time for a 64KB memory block is smaller than for larger block sizes. The reason for this anomaly is the eager data transfer from the CPU. A small block size triggers a higher number of block evictions from the CPU to the accelerator which overlaps with other computations in the CPU. In this benchmark, when the block size goes from 64KB to 128KB, the time required to transfer a block from the CPU to the accelerator becomes longer than the time required by the CPU to initialize the next memory block and, therefore, evictions must wait for the previous transfer to finish before continuing. Hence, a small enough memory block size is essential to overlap data transfers with computations at the CPU.

5.3 Rolling Size

In this experiment we vary the *rolling size* (maximum number of memory blocks on dirty state). The rolling size affects application performance in two different ways. First, the smaller the rolling size, the more eagerly data is transferred to accelerator memory and, thus, the more overlap between data transfers and computation. Second, memory blocks are marked as read-only once they are evicted from the memory block cache. Hence, any subsequent write to any memory location within a evicted block triggers a page fault and the eviction of a block from the memory block cache. This experiment shows that for all Parboil benchmarks, except for *tpacf*, the rolling size does not affect application performance in an appreciable way due to the way they are coded.

Execution time results for the *tpacf* benchmark illustrates quite a pathological case that might be produced by small rolling sizes. Figure 12 shows the execution time of *tpacf* for different block sizes using rolling sizes of 1, 2, and 4. For rolling size values of 1 and 2, and small memory block values, data is being transferred from system memory to accelerator memory continuously. The *tpacf* code initializes shared data structures in several passes. Hence, memory blocks of shared objects are written only once by the CPU before their state is set to read-only and they are transferred to accelerator memory. As the memory block size increases, the cost of data transfers becomes higher and, thus, the execution time increases. When the memory block size reaches a critical value (2MB for *tpacf-2* and 4MB for *tpacf-1*), memory blocks start being overwritten by subsequent passes before they are evicted, which translates to a shorter execution time. Once the complete input data set fits in the rolling size, the execution time decreases abruptly because no unnecessary updates are done. For a rolling size value of 4, the execution time of *tpacf* is almost constant for all block sizes. In this case, data is still being transferred, but the larger number of memory blocks that might be in dirty state allows memory blocks to be written by all passes before being evicted.

This results reveals an important insight that might be especially important for a hardware ADSM implementation. In such an

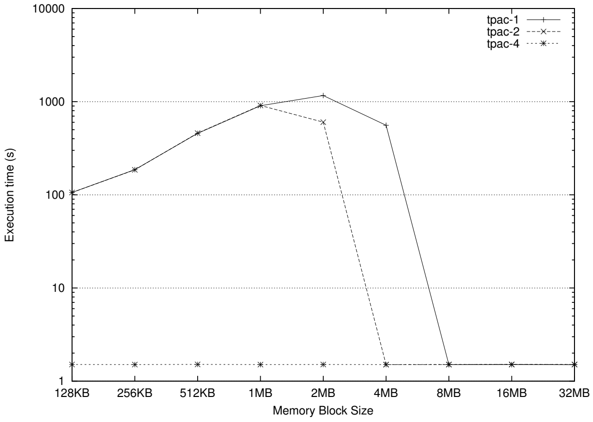


Figure 12. Execution time for *tpacf* using different memory block and rolling sizes

implementation, given a fixed amount of storage, there is a trade-off between the memory block size and the number of blocks that can be stored. Experimental results shows that it is more beneficial to allow a higher number of blocks in dirty state than providing a large block granularity. Other considerations, such different costs for DMA transfers and page faults must be taken into account when designing a hardware ADSM system.

6. Related Work

6.1 Programming Models

Most programming models proposed for massively parallel systems deal with data distribution and kernel scheduling on clusters of computers. Global Arrays [37] provide semantics to divide and access arrays on a distributed memory system. In a data-centric programming model, the accelerator memory hosts all data required by accelerator kernels and, therefore, no data distribution is required if only one accelerator is used for each kernel execution. Global Arrays are compatible with a data-centric programming model and might be used if the execution of a kernel is distributed among several accelerators. ASSIST [42] decomposes programs into components that communicate through data streams. ASSIST requires the programmer to declare modules and connect them using streams. This data-dependence information is used by the ASSIST run-time to schedule the execution of modules on different processors. A data-centric programming model also requires the programmer to assign data structures to computational intensive kernels.

Darlington et al. [14] proposed using *skeleton* functions as part of the programming model. Skeleton functions implement parallel algorithms commonly used in a given class of applications. This approach is, for instance, behind STAPL which provides parallel implementations of STL C++ functions [4]. This approach can be used on top of a data-centric programming model, where the skeleton methods are marked as performance critical and the parallel version is selected to run in an accelerator. An ADSM system can be used to simplify data transfers between the rest of the program and the skeleton functions.

Software development kits for commercially available accelerators such as the Cell Runtime Management Library [25] or NVIDIA CUDA [38], require programmers to explicitly move data between system memory and accelerator memory prior to performing any calculation using these data structures on the accelerator. The OpenCL specification [1] also uses explicit data transfers between the different processors present in the system. The programming model presented in this paper removes the need for explicit data transfers, thus easing application development, and uses CUDA to interact with GPU accelerators. An OpenCL Abstraction Layer is ongoing work and will enable GMAC to be used with a wide variety of accelerators. The programming model offered by the CUBA architecture [19] allows programmers to explicitly select data structures to be hosted in accelerator memory. The model presented in this paper extends CUBA allowing data structures to be assigned to the methods using them. This is a key difference because the programming model used in CUBA requires programmers to statically partition data structures, whereas in a data-centric programming model, data placement is done by the run-time system. CellSS [7] is a programming model where programmers identify tasks and their input and output parameters through source code annotations. The CellSS run-time exploits task-level parallelism by executing independent tasks concurrently. CellSS differs from the data-centric programming model in that CellSS identifies input and output parameters whose value is only known at the method call time, instead of data structures. Hence, the CellSS does not allow data to be eagerly transferred to or from accelerators.

6.2 Distributed Shared Memory

Many hardware and software DSM systems exist, that implement a shared address space on top of physically distributed memories. The ADSM system presented in this paper also provides programmers with the illusion of a shared address space, however, only few processors in the system (i.e., general purpose CPUs) have full access to the whole address space, while other processors (i.e. GPUs or other accelerators) can only access memory that is physically attached to them. This key difference allows ADSM systems to minimize the coherence traffic.

Hardware DSM systems include the necessary logic to implement coherence protocols and detect accesses to shared data. Most hardware DSM systems implement write-invalidate protocols, rely on directories to locate data, and have a cache-line size sharing granularity [16, 18, 20, 33, 36, 44]. There are also hardware implementations that rely on software to implement data replication [10], to support the coherence protocol [2], to virtualize the coherence directory [45], or to select the appropriate coherence protocol [23]. In this work we only explore a software-based implementation of ADSM which implements *release consistency* [33] and uses a larger sharing granularity to minimize the overhead for detection of accesses to shared data structures.

Software DSM systems might be implemented as a compiler extension or as a run-time system. Compiler based DSM systems add semantics to programming languages to declare shared data structures. At compile time, the compiler generates the necessary synchronization and coherence code for each access to any shared data structure [3, 5]. The programming model presented in this paper also requires the programmer to identify those data structures required by performance-critical functions. However, we use a run-time mechanism to detect accesses to shared data structures, since in most cases no coherence or synchronization action is required. Compiler support can be added to our work for identification of data objects modified by kernels and, thus, to avoid unnecessary data transfers.

Run-time DSM implementations provide programmers with the necessary APIs to register shared data structures. A software run-time system uses the memory protection hardware to detect accesses to shared data structures and to perform the necessary coherency and synchronization operations. The run-time system might be implemented as part of the operating system [15, 17] or as an user-level library [8, 12, 31, 34]. The former allows the operating system to better manage system resources (e.g., blocking a process while data is being transfered) but requires a greater implementation effort. User-level DSM libraries require the operating system to bypass and forward protection faults (e.g., as POSIX signals) and to provide system calls to interact with the memory protection hardware. We implement a user-level ADSM library because currently there is no operating system level support for interacting with the CUDA driver. ADSM memory consistency and coherence protocols differ greatly from previous software DSM implementations due to shorter network latencies and different data sharing patterns found in heterogeneous parallel systems as compared with clusters of homogeneous CPU-based computers.

Heterogeneity has been also considered in software DSM systems [9, 47]. These works mainly deal with different endianisms, data type representations and machines running different operating systems. For simplicity, in this paper we assume that processors in the system use same endianism, data representation, and calling conventions. The techniques discussed in [9, 47] are applicable to ADSM whenever processors use different calling conventions or data-type representations.

7. Conclusion

This paper has introduced ADSM, a data-centric programing model, for heterogeneous systems. ADSM presents programmers with a shared address space between general purpose CPUs and accelerators. CPUs can access data hosted by accelerators, but not vice versa. This asymmetry allows memory coherence and consistency actions to be executed solely by the CPU. ADSM exploits this asymmetry (*page faults* triggered only by CPU code), the characteristics of heterogeneous systems (lack of synchronization using program-level shared variables), and the properties of data-centric programming models (*release consistency* and data object access information) to avoid performance bottle-necks traditionally found in symmetric DSM systems.

We have also presented GMAC, a software user-level implementation of ADSM, and discussed the software techniques required to build a shared address space and to deal with I/O and bulk memory in an efficient way using mechanisms offered by most operating systems. We have further presented three different coherence protocols to be used in an ADSM system: batch-update, lazy-update and rolling-update, each one being a refinement of the previous one. Experimental results show that rolling-update and lazy-update greatly reduces the amount of data transferred between system and accelerator memory and performs as well as existent programming models.

Based on our experience with ADSM and GMAC, we argue that future application development for heterogeneous parallel computing systems should use ADSM to reduce the development effort, improve portability, and achieve high performance. We also identify the need for memory virtualization to be implemented by accelerators in order to ADSM systems to be robust for heterogeneous systems containing several accelerators. We have also shown that hardware supported *peer DMA* can increase the performance of certain applications.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625, the FP 7 HiPEAC NoE (ICT-217068). 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 NVIDIA for her hardware donations. Finally, the authors also acknowledge the insightful comments of the GSO and the IMPACT group members.

References

[1] *The OpenCL Specification*, 2009.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine: architecture and performance. In *ISCA '95*, pages 2–13, New York, NY, USA, 1995. ACM.

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Trans. on Computers*, 19(8):26–34, Aug. 1986.

[4] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. *LNCS*, pages 193–208, 2003.

[5] H. Bal and A. Tanenbaum. Distributed programming with shared data. In *ICCL '88*, pages 82–91, Oct 1988.

[6] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC'08*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[7] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *SC'06*, page 86, New York, NY, USA, 2006. ACM.

[8] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *Compcon Spring '93*, pages 528–537, Feb 1993.

[9] R. Bisiani and A. Forin. Multilanguage parallel programming of heterogeneous machines. *IEEE Trans. on Computers*, 37(8):930–945, Aug 1988.

[10] R. Bisiani and M. Ravishankar. PLUS: a distributed shared-memory system. *SIGARCH Comput. Archit. News*, 18(3a):115–124, 1990.

[11] I. Buck. GPU computing with NVIDIA CUDA. In *SIGGRAPH '07*, page 6, New York, NY, USA, 2007. ACM.

[12] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *SOSP '91*, pages 152–164, New York, NY, USA, 1991. ACM.

[13] B.-C. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI '00*, pages 57–69, New York, NY, USA, 2000. ACM.

[14] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE'93*, pages 146–160, London, UK, 1993. Springer-Verlag.

[15] P. Dasgupta, J. LeBlanc, R.J., M. Ahamad, and U. Ramachandran. The clouds distributed operating system. *IEEE Trans. on Computers*, 24(11):34–44, Nov 1991.

[16] G. Delp, A. Sethi, and D. Farber. An analysis of memnet—an experiment in high-speed shared-memory local networking. In *SIGCOMM '88*, pages 165–174, New York, NY, USA, 1988. ACM.

[17] B. Fleisch and G. Popek. Mirage: a coherent distributed shared memory design. In *SOSP '89*, pages 211–223, New York, NY, USA, 1989. ACM.

[18] S. Frank, I. Burkhardt, H., and J. Rothnie. The KSR 1: bridging the gap between shared memory and MPPs. In *Compcon Spring '93*, pages 285–294, Feb 1993.

[19] I. Gelado, J. H. Kelm, S. Ryoo, S. S. Lumetta, N. Navarro, and W. W. Hwu. CUBA: an architecture for efficient cpu/co-processor data communication. In *ICS '08*, pages 299–308, New York, NY, USA, 2008. ACM.

[20] D. B. Gustavson. The scalable coherent interface and related standards projects. *IEEE Micro*, 12(1):10–22, 1992.

[21] S. H. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The chimaera reconfigurable functional unit. *IEEE Trans. on VLSI*, 12(2):206–217, Feb. 2004.

[22] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *FCCM '97*, pages 12–21, Apr 1997.

[23] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The performance impact of flexibility

in the Stanford FLASH multiprocessor. In *ASPLOS '94*, pages 274–285, New York, NY, USA, 1994. ACM.

[24] W. W. Hwu and J. Stone. A programmers view of the new GPU computing capabilities in the Fermi architecture and cuda 3.0. White paper, University of Illinois, 2009.

[25] IBM Staff. *SPE Runtime Management Library*, 2007.

[26] IMPACT Group. Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.

[27] Intel Staff. *Intel 945G Express Chipset Product Brief*, 2005.

[28] Intel Staff. *Intel Xeon Processor 7400 Series Specification*, 2008.

[29] V. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC '09*, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[31] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: distributed shared memory on standard workstations and operating systems. In *WTEC'94*, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[32] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA '09*, pages 140–151, New York, NY, USA, 2009. ACM.

[33] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *ISCA '90*, pages 148–159, New York, NY, USA, 1990. ACM.

[34] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[35] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March-April 2008.

[36] C. Maples and L. Wittie. Merlin: A superglue for multicomputer systems. In *Compcon Spring '90*, volume 90, pages 73–81, 1990.

[37] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *SC'94*, pages 340–349, New York, NY, USA, 1994. ACM.

[38] NVIDIA Staff. *NVIDIA CUDA Programming Guide 2.2*, 2009.

[39] S. Patel and W. W. Hwu. Accelerator architectures. *IEEE Micro*, 28(4):4–12, July-Aug. 2008.

[40] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[41] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. on Computers*, 49(5):465–481, May 2000.

[42] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, 2002.

[43] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The MOLEN polymorphic processor. *IEEE Trans. on Computers*, 53(11):1363–1375, 2004.

[44] D. Warren and S. Haridi. Data Diffusion Machine-a scalable shared virtual memory multiprocessor. In *Fifth Generation Computer Systems 1988*, page 943. Springer-Verlag, 1988.

[45] J. Wilson, A.W., J. LaRowe, R.P., and M. Teller. Hardware assist for distributed shared memory. In *DCS '03*, pages 246–255, May 1993.

[46] Xilinx Staff. *Virtex-5 Family Overview*, Feb 2009.

[47] S. Zhou, M. Stumm, and T. McInerney. Extending distributed shared memory to heterogeneous environments. In *DCS '90*, pages 30–37, May 1990.