FusionSim: Characterizing the Performance Benefits of Fused CPU/GPU Systems

by

Vitaly Zakharenko

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Vitaly Zakharenko 2012
FusionSim: Characterizing the Performance Benefits of Fused CPU/GPU Systems

Vitaly Zakharenko

Master of Applied Science

Graduate Department of Electrical and Computer Engineering
University of Toronto
2012

Abstract

We present FusionSim, a modeling framework capable of cycle-accurate simulation of a complete x86-based computer system with (a) a CPU and a GPU on the same die, and (b) a CPU and a GPU connected as separate components.

We use FusionSim to characterize the performance of the Rodinia benchmarks on fused and discrete systems. We demonstrate that the speed-up due to fusion is highly correlated with the input data size. We demonstrate that for benchmarks that benefit most from fusion, a 9.72x speed-up is possible for small problem sizes. This speedup reduces to 1.84x with medium problem sizes. We study a software-managed coherence solution for the fused system. We find that it imposes a minor performance overhead of 2% for most benchmarks. Finally, we develop an analytical model for the performance benefit that is to be expected from fusion and show that FusionSim follows the predicted performance trend.
Acknowledgments

First and foremost, I would like to thank my advisor Professor Andreas Moshovos for his guidance, inspiration and support. His dedication and expertise has helped me immensely during this work.

I am grateful to my thesis defense committee members for their time and invaluable feedback.

I would like to thank my group members Ioana Burcea, Jason Zebchuk, Kaveh Aasaraai, Maryam Sadooghi-Alvandi, Eslias Ferzli, Ian Katsuno, Goran Narancic, Myrto Papadopoulou, Islam Atta, Alhassan Khedr, Michel Nacouzi, Patrick Judd, Xin Tong and Di Wu for all their help and support.

Last but not least, I would like to thank my family for their support and encouragement throughout my life.
Table of Contents

Acknowledgments........................................................................................................ iii
Table of Contents........................................................................................................ iv
List of Tables ................................................................................................................ vi
List of Figures ............................................................................................................... vii
List of Appendices ...................................................................................................... ix
Chapter 1. Introduction................................................................................................. 1
Chapter 2. FusionSim .................................................................................................. 4
  2.1 FusionSim Simulation Framework .................................................................... 4
  2.2 Discrete FusionSim Simulator .......................................................................... 5
    2.2.1 Structure .................................................................................................... 5
    2.2.2 Start-up and Memory Layout ................................................................... 6
    2.2.3 Main Simulation Loop ............................................................................. 8
    2.2.4 Handling CUDA API Calls ...................................................................... 9
  2.3 Fused FusionSim Simulator ............................................................................. 11
    2.3.1 Structure .................................................................................................. 11
    2.3.2 CPU and GPU Memory Spaces ............................................................... 12
    2.3.3 Memory coherence .................................................................................. 13
    2.3.4 Changes to the CUDA API .................................................................... 14
      2.3.4.1 Discrete System ................................................................................ 15
      2.3.4.2 Fused System .................................................................................. 16
      2.3.4.3 Benchmarks ..................................................................................... 17
      2.3.4.4 Execution Time Measurements ....................................................... 18
      2.3.4.5 Modeling Latencies Associated with CUDA API Calls ................. 18
Chapter 3. Results ...................................................................................................... 20
3.1 Analytical model of performance benefits with Fusion........................................... 20
3.2 Performance with Zero-Overhead Coherence ....................................................... 23
3.3 Performance with Software Coherence .................................................................. 26
3.4 The Effect of the Input Data Size ........................................................................... 27
3.5 Results Analysis..................................................................................................... 29
Chapter 4. Related Work ............................................................................................. 33
Chapter 5. Future work ............................................................................................... 34
Chapter 6. Conclusion ................................................................................................. 35
Chapter 7. References or Bibliography ......................................................................... 36
Appendix A FusionSim Manual...................................................................................... 38
Appendix B Required Benchmark Modifications for execution on the fused system........ 58
List of Tables

Table 1. Discrete FusionSim Specifications .............................................................. 15

Table 2. Fused FusionSim Specifications ................................................................. 16

Table 3. Simulated Rodinia Benchmarks ................................................................. 17

Table 4. Kernel launches and memory transfers of the simulated Rodinia benchmarks .................................................. 18

Table 5. Parameters of the Analytical Model ........................................................ 20
List of Figures

Figure 1. Standard heterogeneous computer system................................................................. 1

Figure 2. Fused heterogeneous computer system. ................................................................. 1

Figure 3. Expanded model of a discrete heterogeneous computer system simulated by Discrete FusionSim .................................................................................................................................... 5

Figure 4. FusionSim’s Memory layout (BM stands for benchmark). ...................................... 6

Figure 5. Single simulation loop of the FusionSim and simulation of the CPU and the GPU clock domains. .................................................................................................................... 8

Figure 6. Illustration of the execution of an example CUDA API function. ....................... 9

Figure 7. Handling of synchronous and asynchronous GPU tasks by the simulator ............ 10

Figure 8. Expanded model of a fused heterogeneous computer system simulated by Fused FusionSim (L2 of the GPU serves as CPU’s L3 cache). ......................................................... 11

Figure 9. Relation between CPU’s virtual address and GPU’s generic virtual address. The yellow sections are mapped to the same physical locations. ......................................................... 13

Figure 10. Speed-up of the Rodinia benchmarks on a fused system with 0.1 µsec kernel spawn latency and no memory coherence overhead. ................................................................................... 24

Figure 11. Speed-up of the Rodinia benchmarks on the fused systems with three different values of kernel spawn latency relative to the optimistic 10 µsec latency discrete system. ................ 25

Figure 12. Overhead of the software-based memory coherence mechanism in terms of benchmark execution time. .............................................................................................................. 26

Figure 13. Speed-up of the Rodinia BFS benchmark on a fused system for different data sizes. The speed-up tends to reduce with the benchmark data input size................................. 27
Figure 14. Speed-up of *Rodinia Gaussian* benchmark on a fused system for different data sizes. The speed-up is inversely proportional to the benchmark data input size. ............................ 28

Figure 15. Kernel throughput of Rodinia benchmarks. ................................................................. 31

Figure 16. Graph of $\frac{\Delta_{\text{TOTAL}}}{\text{data}_{\text{TOTAL}}}$ multiplicand in the speed-up formula (5) normalized to that the value for *backprop*. Note the large value of the expression for *gaus_4* and the small one for *nn*. 32
List of Appendices

1. The website of FusionSim simulation framework (www.fusionsim.ca).


3. Appendix B: Modifications to the benchmark source code for execution on the fused system.
Chapter 1. Introduction

Many modern computer systems are heterogeneous, comprising two very different processing units, the CPU (central processing unit) and the GPU (graphics processing unit), that are both capable of general-purpose computations.

Recently, several non-graphics applications have been designed to partition computations between the CPU and the GPU. Such heterogeneous workloads take advantage of both the high computing power of GPUs for SIMD-friendly multithreaded workloads and the high single-thread performance of CPUs. For many heterogeneous applications, significant performance improvements of up to 1000x have been reported relative to their CPU-only homogeneous counterparts.

Originally, heterogeneous computer systems (see Figure 1) were discrete. That is, the CPU and the GPU were on separate dies, and the CPU DRAM memory and the GPU DRAM memory were private to the CPU and the GPU, respectively. In heterogeneous applications computation processing involves copying data from the CPU’s main memory to the GPU DRAM via a PCIe link, performing the computation on the GPU, and copying the results from the GPU DRAM back to the CPU’s main memory.

Fused heterogeneous architectures (see Figure 2) are now emerging. Commercially available fused systems include Intel’s Sandy Bridge [1] and AMD’s Fusion [2]. In fused systems, both the CPU and the GPU are on the same die and share the main memory DRAM.
The fused architecture promises high performance gains since it alleviates the need to transfer data blocks between distinct DRAM memories thus removing the bandwidth limitations and latency overheads of the PCIe link.

However, the fused design also presents multiple challenges. For example, CPU and GPUs tend to have different memory access patterns and to exhibit different access frequencies challenging existing memory hierarchies. Typically, different processes are employed in the production of CPU and GPU chips, resulting in suboptimal individual performance when fused.

Placing the CPU and the GPU is only the first step in creating a fused system. Several other mechanisms must be introduced or adapted accordingly to achieve the best performance and power. Major components and mechanisms that ought to be studied and adapted include the interconnect, the memory hierarchy, the coherence scheme used to communicate data between the CPU and the GPU, how virtual memory is handled, and how are the GPU and the CPU cores interleaved.

One of the main reasons why the fused architectures have not yet been extensively studied is the lack of open-source simulation tools capable of timing simulation of discrete and fused heterogeneous. In this paper, we present FusionSim [4], an open-source modeling framework capable of cycle-accurate simulation of a complete x86-based computer system with a) a CPU and a GPU on the same die (fused), and b) a CPU and a GPU connected as separate components (discrete).

We use FusionSim to demonstrate the potential benefits of fused systems. To do so, we model a discrete system that is representative of a commercially available solution and a hypothetical, realistic fused system. Our simulated, hypothetical fused system has partially private CPU and GPU memory hierarchies, a shared last level cache, and a common main memory. It uses a software-managed coherence mechanism for correct CPU and GPU communication.

Note that FusionSim was not intended to model an AMD Fusion series system. It models a hypothetical fused system.

Using workloads from the Rodinia benchmark suite [3] we study the relative performance of the two systems. We demonstrate that the speed-up due to fusion depends primarily on the input data
size. The larger the input data size, the less beneficial fusion is in terms of performance. We provide a theoretical explanation of the experimental simulation results obtained with FusionSim. We show that our simple, software-managed coherence mechanism is of low overhead for most workloads.

To the best of our knowledge FusionSim, is the first open-source simulation framework that can simulate fused and discrete CPU and GPU systems. It can serve as the building block upon which further work can investigate alternative fused organizations and mechanisms.
Chapter 2. FusionSim

2.1 FusionSim Simulation Framework

There are two versions of FusionSim, the discrete and fused versions. Both FusionSim simulator versions model an x86 out-of-order CPU and a CUDA-capable GPU that operate concurrently. For all system components, the simulation framework provides detailed timing models that were available. It accurately models modern commercial hardware.

FusionSim correctly models ordering and overlap in time of asynchronous and synchronous operations, memory transfers, CUDA events, kernels and CPU processing.

The simulator combines and modifies two major open-source simulators, PTLsim [5] modeling the CPU and GPGPU-Sim [13] modeling the GPU. Additionally, FusionSim uses a modified implementation of the cache and memory hierarchy originating from the MARSSx86 [6] x86 full-system simulator. FusionSim also uses the Intersim Network-on-Chip simulator, a part of GPGPU-Sim originating from BookSim [7].

The rest of this chapter describes first the discrete version and later the fused version of FusionSim.
2.2 Discrete FusionSim Simulator

2.2.1 Structure

![Diagram of discrete heterogeneous computer system](image)

**Figure 3.** Expanded model of a discrete heterogeneous computer system simulated by Discrete FusionSim

**Figure 3** shows how different parts of the discrete heterogeneous system are modeled by the Discrete FusionSim simulator version. The simulator comprises models for a CPU, a CPU cache hierarchy, a PCIe link, a GPU, a GPU cache hierarchy and interconnect, and CPU and GPU DRAMs.

*User-space PTLsim [5]* simulator is used to model the CPU. *User-space PTLsim* is a cycle-accurate, fast and detailed out-of-order x86 CPU simulator that accurately models user-space code and relies on the underlying Linux OS for modelling of system calls. Currently, FusionSim supports only one CPU core. The CPU-side cache hierarchy of FusionSim is extracted from MARSSx86 [6]. The cache system is modular, configurable and detailed. The main memory is modeled by a banked DRAM memory model also originating from MARSSx86. The PCIe link is modelled as a channel of configurable bandwidth and latency. The GPU is modeled by GPGPU-Sim [14], a cycle-level GPU performance simulator that achieves very high correlation vs. modern commercially available NVidia GPUs.
2.2.2 Start-up and Memory Layout

Besides the configuration files the simulator requires only one input: the unmodified standard Linux executable of the benchmark.

Building upon PTLSim’s method, FusionSim creates a new benchmark process and injects itself into the virtual memory space of the benchmark process. Further, the simulator sets up its own private stack and private heap memory managed by PTLSim’s private dynamic memory management mechanism. FusionSim (including stack, heap, code and data of FusionSim) remains invisible to the benchmark and inaccessible by it.

Once the simulator injection is finished and all simulation structures are set up, the virtual x86 PTLsim CPU starts executing the x86 code of the benchmark starting from the code of the dynamic loader (also known as Linux interpreter). Once the execution of the code of the dynamic loader is finished, the memory layout of the process is as shown in Figure 4.

The dynamically-linked CUDA benchmark is then linked, at runtime with the CUDA dynamic library libcudart.so. FusionSim implements its own libcudart.so library that mimics the original dynamic library API. At the beginning of the simulation this substitute library is loaded into the virtual space of the benchmark process together with all the other dynamic libraries referenced by the benchmark executable.

Figure 4. FusionSim’s Memory layout (BM stands for benchmark).
FusionSim reserves one page at a fixed address in the virtual memory space of the benchmark process that serves for communication between FusionSim and the simulated benchmark. The page serves for signaling of pending CUDA API calls to FusionSim, for the transfer of API call arguments from the benchmark to FusionSim and for the transfer of return values from FusionSim to the benchmark.

The replacement libcudart.so library implements light-weight substitutes of all CUDA API functions. The substitute functions store arguments of the API call to a structure within the shared page and post a POSIX semaphore (by calling `sem_post` function of POSIX `pthread` library) within the same page to signal presence of a pending CUDA API call to the simulator.
• Single simulation loop:
  - Each loop cycle == tick of a **virtual** ‘common’ clock
  - \( x \) GPU_MULTIPLIER = GPU_FREQ
  - \( x \) CPU_MULTIPLIER = CPU_FREQ

  **WHILE** (1) {
    **FOR** GPU_MULTIPLIER **ITERATIONS DO** {
      GPU_CYCLE()
    }
    **FOR** CPU_MULTIPLIER **ITERATIONS DO** {
      CPU_CYCLE()
    }
  }

**Figure 5.** Single simulation loop of the FusionSim and simulation of the CPU and the GPU clock domains.

### 2.2.3 Main Simulation Loop

Simulation of both the CPU and the GPU is performed in a single simulation loop. FusionSim defines a virtual common clock of a configurable frequency. The frequencies of the CPU and the GPU are derived from the common clock frequency using configurable CPU and GPU frequency multipliers.

On each iteration of the common clock the CPU and the GPU are advanced by a number of cycles equal to the CPU frequency multiplier and the GPU frequency multiplier, respectively.
2.2.4 Handling CUDA API Calls

When the execution path of the simulated CUDA benchmark reaches an API function, the virtual CPU of PTLsim executes a light-weight function implemented in the FusionSim’s substitute of the libcudart.so library. **Figure 6** shows an example how an API function `cudaMemcpyAsync` is implemented in the replacement library. The function stores the API call arguments to a shared structure allocated in the dedicated page for sharing data between the simulated benchmark and FusionSim. Then it posts the shared semaphore within that structure to signal presence of a pending API call to FusionSim. All these actions are performed by the simulated CPU.

---

**Figure 6.** Illustration of the execution of an example CUDA API function.
FusionSim

Figure 7. Handling of synchronous and asynchronous GPU tasks by the simulator.

On each GPU cycle, FusionSim polls the shared semaphore to determine whether there is a pending CUDA API call that needs to be processed. If FusionSim identifies presence of a pending call it enqueues the GPU task associated with the CUDA call into the corresponding CUDA stream.

There are two types of CUDA API calls: synchronous and asynchronous. The synchronous calls (also known as blocking calls) return only after the associated GPU task has been completed. On the other hand, the asynchronous calls (also known as non-blocking calls) return immediately after enqueuing the GPU task for future processing.

Depending on the kind of the CUDA call, synchronous or asynchronous, and the state of the GPU (depends on the status of the preceding API calls) the simulator either unlocks the semaphore or keeps the semaphore locked until a later GPU cycle when the blocking task (not necessarily the current task) finishes (see Figure 7).

The virtual CPU of PTLsim also polls the semaphore in a loop. In practice, this loop iterates only a few times unless the API call is recognized by FusionSim as synchronous. Once the polling determines that the semaphore is unlocked the benchmark proceeds on its execution path on the simulated CPU.
2.3 Fused FusionSim Simulator

2.3.1 Structure

**Figure 8** illustrates how different parts of the discrete heterogeneous system are modeled by the Fused FusionSim simulator version. The Fused FusionSim replaces one of the Processing Clusters of GPGPU-Sim with a block including the PTLsim CPU core and its CPU core cache hierarchy. As in the Discrete FusionSim version, the Fused FusionSim uses a cache hierarchy derived from MARSSx86.

The CPU cache hierarchy interfaces with Intersim, a NoC model of GPGPU-Sim. Intersim is a modified version of BookSim [7]. Intersim connects the CPU cache hierarchy Processing Clusters with the shared last level cache (LLC) which is tiled. The LLC is modeled using the L2 cache model of GPGPU-Sim. In the default configuration, the LLC appears as an L2 to the GPU clusters and as an L3 to the CPU clusters. The LLC caches main memory data of the CPU and blocks from the global and the local CUDA memory spaces of the GPU. By default, the size of the LLC is increased to accommodate caching of the CPU memory accesses.

The L1 and L2 caches in the CPU cache hierarchy as well as the L1 caches in the GPU cache hierarchy remain private to the CPU and the GPU, respectively. Sharing of the DRAM by the CPU and the GPU alleviates the need for memory transfers between the CPU and the GPU.
2.3.2 CPU and GPU Memory Spaces

The CUDA specification defines different memory spaces accessible by the GPU in a discrete heterogeneous system (see [16]): the register space, the shared memory space, the local memory space, the global memory space, the constant memory space and the texture memory space. Among these memory spaces, the register space, the shared memory space and the local memory space are accessible by the GPU only. On the other hand, the global, constant and texture memory spaces are accessible by both the CPU and the GPU and are mapped to the DRAM on the GPU card.

In the modeled fused system the global, constant and texture memory are mapped to the same physical DRAM and shared by the CPU and the GPU. Other CUDA memory spaces including the CUDA local space and the CUDA shared space must remain private to the GPU and inaccessible by the CPU. This represents a challenge for a fused system that uses a common memory system for all address spaces. Specifically, the DRAM memory stores memory locations associated with both CUDA global space (shared between the CPU and the GPU) and CUDA local space (private to the GPU). This section explains the method we used to simulate these distinct address spaces using one memory space thus avoiding the collisions among them. This is necessary to simulate CUDA benchmarks that use distinct memory spaces.

Both the CPU and the GPU implement virtual memory. We use the term generic virtual address space as the virtual address space used by the GPU.

The upper 3 GBytes of the generic virtual memory space are mapped to the CUDA global space which is shared between the CPU and the GPU. All virtual addresses by GPU access memory without having their address modified. However, FusionSim modifies all CPU access so that the space it see represents a 1GByte-shifted version of the GPU virtual address. Specifically, the CPU virtual address X maps to the same physical location as the generic virtual address (X + 0x40000000).
Since the upper 1 GByte of the CPU virtual memory is only used by the Linux kernel and the user-space PTLsim only simulates the user-space code, this guarantees that FusionSim will never generate memory accesses to CPU virtual addresses in the top 1GByte of the 32-bit address space. Therefore, the 1GByte-shifted CPU virtual memory space will still fit all the simulated memory accesses within the 32-bit generic virtual address space.

The lower 1 GByte of the generic virtual memory space is reserved and divided among all CUDA address spaces that are private to the GPU and inaccessible by the CPU. These are the local CUDA address space and the shared CUDA address space. Figure 9 illustrates the structure and the relation between the CPU virtual and the generic virtual memory spaces. The light grey sections are mapped to the same physical location.

The memory requests by the CPU get their virtual address translated to the generic virtual address, which is subsequently translated to the physical address by the MMU. The memory requests by the GPU are directly translated to the physical address by the MMU. The caches are all physically-addressed. The MMU itself is supposed to be shared between the CPU and the GPU. The MMU is not modeled by the FusionSim, resulting in generic virtual addresses being used as the physical addresses.

2.3.3 Memory coherence

Since both the private L1/L2 caches of the CPU, and the private L1 caches of the GPU, cache blocks from the global CUDA address space it is possible that a block may be cached simultaneously in the CPU-private and the GPU-private caches. Such blocks may contain different data as writes by the CPU or the GPU are not coherent with respect to each other.

Figure 9. Relation between CPU’s virtual address and GPU’s generic virtual address. The yellow sections are mapped to the same physical locations.
A key architectural decision is how to enforce coherence across the CPU and GPU. In this work we study a simple software-based solution. Future work can investigate other solutions. We extend the CUDA API with one function, `cudaSelectivelyFlush`. For each memory structure residing in the global CUDA address space (shared by the CPU and the GPU) and accessed by a kernel (for either read or write), the programmer is required to add a call to `cudaSelectivelyFlush` specifying the memory address and the size of the data structure. The new API function generates memory requests that flush modified (and invalidate unmodified) cache lines within the specified memory regions from the private CPU L1 and L2 caches to the last level cache and the DRAM, which are both common to the CPU and the GPU.

The above software-based coherence solution updates the LLC before the kernel starts logically transferring the ownership of these memory regions to the GPU. When the GPU kernel finishes the GPU flushes all its L1 caches before signaling kernel completion to the CPU. This logically transfers the ownership of the memory regions to the CPU.

### 2.3.4 Changes to the CUDA API

Since the main memory is shared between the CPU and the GPU, no CUDA API functions are required for the transfer of data blocks between the main memory and the private DRAM of the GPU. Thus, all `cudaMemcpy` variants and `cudaMemset` are eliminated. Additionally, since there is no separate device DRAM memory, the CUDA API functions responsible for device memory allocation (all variants of `cudaMalloc` and `cudaFree`) are also eliminated, unless they are for private GPU buffers. The software-based coherence mechanism requires one additional CUDA API function, `cudaSelectivelyFlush`, as specified in the previous section.

Appendix B appendix provides an example of modifications to the benchmark source code required for execution on the fused heterogeneous system.
2.3.4.1 Discrete System

Table 1. Discrete FusionSim Specifications.

<table>
<thead>
<tr>
<th>GPGPU-Sim GPU Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># Streaming Multiprocessors</td>
<td>27</td>
</tr>
<tr>
<td>Warp Size</td>
<td>32</td>
</tr>
<tr>
<td>SIMD Pipeline Width</td>
<td>8</td>
</tr>
<tr>
<td>Number of Threads / Core</td>
<td>1024</td>
</tr>
<tr>
<td>Number of Registers / Core</td>
<td>16384</td>
</tr>
<tr>
<td>Shared Memory / Core</td>
<td>16KB</td>
</tr>
<tr>
<td>Constant Cache Size / Core</td>
<td>8KB, 64B line, 2-way assoc.</td>
</tr>
<tr>
<td>Texture Cache Size / Core</td>
<td>5KB, 128B line, 5-way assoc.</td>
</tr>
<tr>
<td>Number of Memory Channels</td>
<td>8</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>16KB, 64B line, 8-way assoc.</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>128KB, 64B line, 8-way assoc.</td>
</tr>
<tr>
<td>Compute Core Clock</td>
<td>1300 MHz</td>
</tr>
<tr>
<td>Interconnect Clock</td>
<td>650 MHz</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>800 MHz</td>
</tr>
<tr>
<td>DRAM request queue capacity</td>
<td>16</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>Out of Order (FR-FCFS)</td>
</tr>
<tr>
<td>Branch Divergence Method</td>
<td>PDOM [8]</td>
</tr>
<tr>
<td>Warp Scheduling Policy</td>
<td>Loose Round Robin</td>
</tr>
<tr>
<td>GDDR3 Memory Timing</td>
<td>tCL=10 tRP =10 tRC =35</td>
</tr>
<tr>
<td></td>
<td>tRAS =25 tRCD =12 tRRD =8</td>
</tr>
<tr>
<td>Memory Channel BW</td>
<td>8 (Bytes/Cycle)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTLSim CPU Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Core type</td>
<td>Out-of-Order</td>
</tr>
<tr>
<td>Core Frequency</td>
<td>3.25GHz</td>
</tr>
<tr>
<td>L1 I cache: 256 sets, 8-way SA, 64 B Block Size, 2 cycle latency, 1 read and 1 write ports</td>
<td></td>
</tr>
<tr>
<td>L1 D cache: 256 sets, 8-way SA, 64 B Block Size, 2 cycle latency, 2 read and 1 write ports</td>
<td></td>
</tr>
<tr>
<td>L2 Cache: 4096 sets, 8-way SA, 64B blocks, 5 cycle latency, 2 write ports, 2 read ports</td>
<td></td>
</tr>
<tr>
<td>L3 Cache: 4096 sets, 8-way SA, 64B blocks, 8 cycle latency, 2 read and 2 write ports</td>
<td></td>
</tr>
<tr>
<td>Main Memory: 64 outstanding requests, 8 banks, 160 cycle latency</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUDA Stream Operation Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BW of memcpy_d2d</td>
<td>80 GBytes/sec</td>
</tr>
<tr>
<td>BW of memcpy_d2h, memcpy_h2d</td>
<td>5 GBytes/sec</td>
</tr>
<tr>
<td>Latency of memcpy_d2d</td>
<td>1.2 µsec</td>
</tr>
<tr>
<td>Latency of memcpy_d2h, memcpy_h2d</td>
<td>7.5 µsec</td>
</tr>
<tr>
<td>memset</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 details the discrete system we modeled.

We model a single-core out-of-order CPU operating at 3.25 GHz with private L1 data and instruction caches, and unified (i.e., caching both data and instruction accesses) L2 and L3 caches. All the caches are write-back. The GPU side is modeled after NVIDIA’s Quadro FX 5800 (GT200) with slightly modified caches. A simple bandwidth/latency models the delays of CUDA stream operations transferring data between the main memory and the GPU DRAM memory.
2.3.4.2 Fused System

As Table 2 shows, we model a single-core out-of-order CPU operating at 3.25 GHz frequency with private L1 and L2 caches. The CPU uses the L2 cache of the GPU as its L3 cache. All the caches are write-back. Our GPU model is based on NVIDIA's Quadro FX 5800 (GT200) with the total size of L2 cache increased to 2 MB and configured as read/write data cache.

Table 2. Fused FusionSim Specifications

<table>
<thead>
<tr>
<th>GPGPU-SIM GPU Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># Streaming Multiprocessors</td>
<td>27</td>
</tr>
<tr>
<td>Warp Size</td>
<td>32</td>
</tr>
<tr>
<td>SIMD Pipeline Width</td>
<td>8</td>
</tr>
<tr>
<td>Number of Threads / Core</td>
<td>1024</td>
</tr>
<tr>
<td>Number of Registers / Core</td>
<td>16384</td>
</tr>
<tr>
<td>Shared Memory / Core</td>
<td>16KB</td>
</tr>
<tr>
<td>Constant Cache Size / Core</td>
<td>8KB, 64B line, 2-way assoc.</td>
</tr>
<tr>
<td>Texture Cache Size / Core</td>
<td>5KB, 128B line, 5-way assoc.</td>
</tr>
<tr>
<td>Number of Memory Channels</td>
<td>8</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>16KB, 64B line, 8-way assoc.</td>
</tr>
<tr>
<td>Last Level Cache</td>
<td>2MB, 64B line, 8-way assoc.,</td>
</tr>
<tr>
<td>Compute Core Clock</td>
<td>1300 MHz</td>
</tr>
<tr>
<td>Interconnect Clock</td>
<td>650 MHz</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>800 MHz</td>
</tr>
<tr>
<td>DRAM request queue capacity</td>
<td>16</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>Out of Order (FR-FCFS)</td>
</tr>
<tr>
<td>Branch Divergence Method</td>
<td>PDOM [8]</td>
</tr>
<tr>
<td>Warp Scheduling Policy</td>
<td>Loose Round Robin</td>
</tr>
<tr>
<td>GDDR3 Memory Timing</td>
<td>tCL=10 tRP =10 tRC =35</td>
</tr>
<tr>
<td></td>
<td>tRAS =25 tRCD =12 tRRD =8</td>
</tr>
<tr>
<td>Memory Channel BW</td>
<td>8 (Bytes/Cycle)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTLSim CPU Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Core type</td>
<td>Out-of-Order</td>
</tr>
<tr>
<td>Core Frequency</td>
<td>3.25GHz</td>
</tr>
<tr>
<td>L1 I cache</td>
<td>256 sets, 8-way SA, 64 B Block Size, 2 cycle latency, 1 read and 1 write port</td>
</tr>
<tr>
<td>L1 D cache</td>
<td>256 sets, 8-way SA, 64 B Block Size, 2 cycle latency, 2 read and 1 write ports</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>4096 sets, 8-way SA, 64B blocks, 5 cycle latency, 2 write ports, 2 read ports</td>
</tr>
</tbody>
</table>


2.3.4.3 Benchmarks

We simulate ten benchmarks from the Rodinia [3] benchmark suite with the input data sets shown in Table 3.

Table 3 reports, for each benchmark, the number of kernel invocations, memory copy calls and the total amount of data transferred from/to the GPU.


Table 3. Simulated Rodinia Benchmarks

<table>
<thead>
<tr>
<th>Short Benchmark name</th>
<th>Data input arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfs_small</td>
<td>graph4096.txt</td>
</tr>
<tr>
<td>gaus_256, gaus_128, gaus_64, gaus_32, gaus_16, gaus_4</td>
<td>matrix256.txt, matrix128.txt, matrix64.txt, matrix32.txt, matrix16.txt, matrix4.txt</td>
</tr>
<tr>
<td>hotspot</td>
<td>64 2 2 temp_64 power_64</td>
</tr>
<tr>
<td>lavaMD</td>
<td>-boxes1d 1</td>
</tr>
<tr>
<td>lud</td>
<td>64.dat</td>
</tr>
<tr>
<td>nn</td>
<td>filelist_1 -r 5 -lat 30 -lng 90</td>
</tr>
<tr>
<td>nw</td>
<td>256 10</td>
</tr>
<tr>
<td>pathfinder</td>
<td>10 100 20</td>
</tr>
<tr>
<td>srad</td>
<td>64 64 0 127 0 127 0.5 2</td>
</tr>
<tr>
<td>bfs_large</td>
<td>graphs65536.txt</td>
</tr>
<tr>
<td>backprop</td>
<td>1024</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Short Name</th>
<th>Full Name</th>
<th>Dwarves</th>
<th>Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>lud</td>
<td>lud</td>
<td>LU Decomposition</td>
<td>Dense Linear Algebra</td>
<td>Linear Algebra</td>
</tr>
<tr>
<td>hot</td>
<td>hot</td>
<td>HotSpot</td>
<td>Structured Grid</td>
<td>Physics Simulation</td>
</tr>
<tr>
<td>backprop</td>
<td>backprop</td>
<td>Back Propagation</td>
<td>Unstructured Grid</td>
<td>Pattern Recognition</td>
</tr>
<tr>
<td>nw</td>
<td>nw</td>
<td>Needleman-Wunsch</td>
<td>Dynamic Programming</td>
<td>Bioinformatics</td>
</tr>
<tr>
<td>bfs</td>
<td>bfs</td>
<td>Breadth-First Search1</td>
<td>GraphTraversal</td>
<td>Graph Algorithms</td>
</tr>
<tr>
<td>srad</td>
<td>srad</td>
<td>SRAD</td>
<td>Structured Grid</td>
<td>Image Processing</td>
</tr>
<tr>
<td>pathfinder</td>
<td>pathfinder</td>
<td>PathFinder</td>
<td>Dynamic Programming</td>
<td>Grid Traversal</td>
</tr>
<tr>
<td>gaus</td>
<td>gaus</td>
<td>Gaussian Elimination</td>
<td>Dense Linear Algebra</td>
<td>Linear Algebra</td>
</tr>
<tr>
<td>nn</td>
<td>nn</td>
<td>k-Nearest Neighbors</td>
<td>Dense Linear Algebra</td>
<td>Data Mining</td>
</tr>
<tr>
<td>lavaMD</td>
<td>lavaMD</td>
<td>LavaMD2</td>
<td>Structured Grid</td>
<td>Molecular Dynamics</td>
</tr>
</tbody>
</table>
### Table 4. Kernel launches and memory transfers of the simulated Rodinia benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Kernel Invocations</th>
<th>Total data transferred from the host to the device</th>
<th>Number of host-to-device memcpy invocations</th>
<th>Total data transferred from the device to the host</th>
<th>Number of device-to-host memcpy invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>backprop</td>
<td>2</td>
<td>213268</td>
<td>5</td>
<td>77896</td>
<td>3</td>
</tr>
<tr>
<td>bfs_large</td>
<td>20</td>
<td>2555914</td>
<td>16</td>
<td>262154</td>
<td>11</td>
</tr>
<tr>
<td>bfs_small</td>
<td>16</td>
<td>159752</td>
<td>14</td>
<td>16392</td>
<td>9</td>
</tr>
<tr>
<td>gauss_256</td>
<td>510</td>
<td>525312</td>
<td>3</td>
<td>525312</td>
<td>3</td>
</tr>
<tr>
<td>gauss_128</td>
<td>254</td>
<td>131584</td>
<td>3</td>
<td>131584</td>
<td>3</td>
</tr>
<tr>
<td>gauss_64</td>
<td>126</td>
<td>33024</td>
<td>3</td>
<td>33024</td>
<td>3</td>
</tr>
<tr>
<td>gauss_32</td>
<td>62</td>
<td>8320</td>
<td>3</td>
<td>8320</td>
<td>3</td>
</tr>
<tr>
<td>gauss_16</td>
<td>30</td>
<td>2112</td>
<td>3</td>
<td>2112</td>
<td>3</td>
</tr>
<tr>
<td>gauss_4</td>
<td>6</td>
<td>144</td>
<td>3</td>
<td>144</td>
<td>3</td>
</tr>
<tr>
<td>hotspot</td>
<td>1</td>
<td>32768</td>
<td>2</td>
<td>16384</td>
<td>1</td>
</tr>
<tr>
<td>lavaMD</td>
<td>1</td>
<td>7744</td>
<td>4</td>
<td>3200</td>
<td>1</td>
</tr>
<tr>
<td>lud</td>
<td>10</td>
<td>16384</td>
<td>1</td>
<td>16384</td>
<td>1</td>
</tr>
<tr>
<td>nn</td>
<td>1</td>
<td>85528</td>
<td>1</td>
<td>42764</td>
<td>1</td>
</tr>
<tr>
<td>nw</td>
<td>31</td>
<td>528392</td>
<td>2</td>
<td>264196</td>
<td>1</td>
</tr>
<tr>
<td>pathfinder</td>
<td>50</td>
<td>40000</td>
<td>2</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>srad</td>
<td>4</td>
<td>32768</td>
<td>2</td>
<td>32768</td>
<td>2</td>
</tr>
</tbody>
</table>

We use the unmodified Rodinia benchmarks for simulations on the discrete system. For simulations on the fused system, we modify the Rodinia benchmarks to exclude CUDA API calls responsible for memory transfers between the GPU DRAM and the main memory. We also exclude CUDA API calls for allocation on the GPU DRAM memory.

#### 2.3.4.4 Execution Time Measurements

In our measurements of execution time, we exclude the time required for data input generation (e.g., reading from a file or random input generation) and output of the results (e.g., writing to screen or to a file).

#### 2.3.4.5 Modeling Latencies Associated with CUDA API Calls

Fusion may impact the latency of most CUDA API calls. To understand the performance potential of reduced API latencies and to appropriately model the latency of the CUDA calls we measured their latency on existing systems and then studied a range of latencies based on these measurements.
Kernel spawn latency is the delay between the `cudaLaunchKernel` API function call and the actual kernel execution on the GPU. To measure the kernel spawn latency we modified the `bandwidthTest` utility from the CUDA SDK to measure the throughput of a microbenchmark kernel and estimated the kernel spawn latency on a number of actual discrete systems. Our measurements indicate kernel spawn latencies between 10 µsec and 100 µsec. We use these two values to model two discrete systems, the optimistic and the pessimistic discrete system, respectively. Since we expect the kernel spawn latency to be reduced with fusion and we modeled a kernel spawn latencies of 0.1 µsec (≈300 CPU cycles), 1 µsec and 10 µsec.

Using the original `bandwidthTest` utility we measured the latency of the CUDA memory copy operations on an actual discrete system and appropriately configured the Discrete FusionSim model.

The fused simulator appropriately models the latency of flushing data from the private caches. This flushing replaces the memory copy operations of discrete systems.
Chapter 3. Results

In Section 3.1, we develop an analytical model for the expected performance benefits from fusion and identify some application characteristics that favour fusion. The rest of the sections present experimental results that confirm most of the conclusions drawn based on the analytical model. In Section 3.2, we compare the performance of fused and discrete systems ignoring the overhead of our software-based coherence mechanism. We study the effect of the CUDA kernel spawn latency on the discrete system demonstrating its impact on the benefit from fusion. In Section 3.3, we measure performance losses incurred by the coherence mechanism. Section 3.4 demonstrates how the performance boost from fusion varies as a function of the input data set size. Section 3.5 uses the analytical model to explain the experimental speed-up results for benchmarks that benefit the most and the least from fusion.

3.1 Analytical model of performance benefits with Fusion

In this section, we develop an analytical model for the expected performance benefits from fusion. We demonstrate analytically that the benefits from fusion are higher for benchmarks that a) operate on small data inputs; b) implement kernels of high data throughput; c) do multiple kernel launch and memory copy CUDA API calls and d) spend most of their time executing the CUDA code. Additionally, we demonstrate analytically that the speed-up from the CPU/GPU fusion reduces with the input data size and becomes insignificant for large input data sizes.

Table 5. Parameters of the Analytical Model

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_{GPU}$</td>
<td>Speed-up of the CUDA portion of the benchmark due to fusion.</td>
</tr>
<tr>
<td>$g_{rot}$</td>
<td>Total speed-up of the benchmark</td>
</tr>
<tr>
<td>$n_{ks}$</td>
<td>Number of kernel invocations</td>
</tr>
<tr>
<td>$t_{GPU}$</td>
<td>Time consumed by execution of the CUDA code</td>
</tr>
<tr>
<td>$data_{rot}$</td>
<td>Total input data size of the benchmark</td>
</tr>
<tr>
<td>$data_{ks}$</td>
<td>Data size per kernel invocation</td>
</tr>
<tr>
<td>$\delta_{rot}$</td>
<td>Total latency of a single computation iteration</td>
</tr>
<tr>
<td>$\delta_{ks}$</td>
<td>Kernel spawn latency</td>
</tr>
<tr>
<td>$\delta_{copy}$</td>
<td>Latency of the CUDA host-to-device and device-to-host memory</td>
</tr>
<tr>
<td>$\Theta_{ks}$</td>
<td>Kernel throughput</td>
</tr>
<tr>
<td>$\Theta_{copy}$</td>
<td>Bandwidth of the host-to-device and device-to-host copy</td>
</tr>
</tbody>
</table>
**Table 5** lists the various parameters used in the development of the analytical model of this section. The kernel can be modeled as a channel of throughput $\Theta_{ker}(data_{ker})$ as the actual throughput will vary depending on $data_{ker}$. As $data_{ker}$ increases, $\Theta_{ker}$ saturates. For simplicity, and without the loss of generality, in the rest of the section we will simply write $\Theta_{ker}$.

First, we develop an expression for the time $t_{GPU}$ spent in execution of the CUDA code on a discrete system.

Most of existing CUDA applications (including all the considered Rodinia benchmarks) exhibit the following computation pattern:

```
For $n_{ker}$ iterations do:
  1. Copy the input data from the host to the device
  2. Launch kernel on the data
  3. Copy the results from the device to the host
```

For such applications $t_{GPU}$ is described by the following:

$$
t_{GPU} = n_{ker} \times \left( \delta_{tot} \times \frac{data_{ker}}{\Theta_{ker}} \right)
$$

where $\Theta_{ker}$ is the kernel data throughput and $\delta_{tot}$ is the total latency per iteration resulting from both the memory transfers and the kernel spawn.

For CUDA applications that do not utilize multiple asynchronous concurrent CUDA streams (see [16]) the total latency per single computation iteration $\delta_{tot}$ is comprised of the time spent transferring the data to or from the device and the kernel spawn latency:

$$
\delta_{tot} = 2 \times \left( \delta_{copy} \times \frac{data_{ker}}{\Theta_{copy}} \right) + \delta_{ks}
$$

(2)

The above expression holds true for all the considered Rodinia benchmarks.

Since on fused systems this latency reduces to $\delta_{tot}' = \delta_{ks}' \leq \delta_{ks}$, the time $t_{GPU}'$ of executing the CUDA code on the fused system is given by

$$
t_{GPU}' = n_{ker} \times \left( \delta_{ks}' \times \frac{data_{ker}}{\Theta_{ker}} \right) \approx n_{ker} \times \frac{data_{ker}}{\Theta_{ker}}
$$

(3)

Since the kernel spawn latency is insignificant compared to the time required for input data
processing, the speed-up of the CUDA code is accurately approximated by

\[ G_{\text{GPU}} = \frac{t_{\text{GPU}}}{t'_{\text{GPU}}} \approx \frac{n_{\text{ker}}}{\Delta_{\text{ker}} \times \Theta_{\text{ker}}} + 1 \quad (4) \]

Since \( n_{\text{ker}} = \frac{\Delta_{\text{ker}}}{\Theta_{\text{ker}}}, \) we obtain

\[
G_{\text{GPU}} \approx \frac{\delta_{\text{tot}} \times n_{\text{ker}}}{\Delta_{\text{tot}}} \times \Theta_{\text{ker}} + 1 = 1 + \frac{\Delta_{\text{total}}}{\Delta_{\text{total}}} \times \Theta_{\text{kernel}} + \Theta_{\text{kernel}} \Theta_{\text{copy}} \quad (5)
\]

Here \( \Delta_{\text{TOTAL}} \) is the total latency accumulated during the benchmark execution and comprising all kernel spawn and memory transfer latencies.

The 2\textsuperscript{nd} term of (5) is responsible for the speed-up due to the reduction of the total cumulative latency \( \Delta_{\text{TOTAL}} \) by fusion. Intuitively, this term is the number of times the GPU computation task could have been accomplished during the total latency time of \( \Delta_{\text{TOTAL}} \) assuming infinite PCIe bandwidth. Please note that \( \Delta_{\text{TOTAL}} \) is only reduced (and is not entirely eliminated) by fusion since the kernel spawn latency persists in the fused system (it is reduced due to the tighter CPU/GPU integration, however it is not entirely eliminated).

The 3\textsuperscript{rd} term of (5) is responsible for the speed-up due to the elimination of the PCIe throughput limitation. Intuitively, this term is the number of times the GPU computation throughput is faster than the PCIe memory transfer throughput.

Equation (5) can be rewritten as follows:

\[ G_{\text{GPU}} \approx 1 + \frac{\delta_{\text{tot}} \times \Theta_{\text{kernel}} (\text{data}_{\text{ker}})}{\text{data}_{\text{ker}}} + \frac{\Theta_{\text{kernel}} (\text{data}_{\text{ker}})}{\Theta_{\text{copy}}} \quad (6) \]

Please also note that the throughput \( \Theta_{\text{ker}} \) of a benchmark kernel increases with \( \text{data}_{\text{ker}} \) for small \( \text{data}_{\text{ker}} \) values and saturates to a constant for large \( \text{data}_{\text{ker}} \) values. The throughput saturates when the input data size is sufficient for maximum possible warp scheduler occupancy for the given benchmark kernel.

Due to the \( \Theta_{\text{kernel}} (\text{data}_{\text{ker}}) \) dependency and since \( \text{data}_{\text{ker}} \) is in the denominator of the 2\textsuperscript{nd} term in (6) it follows that (a) the 2\textsuperscript{nd} term is significant and predominant for small \( \text{data}_{\text{ker}} \) and is insignificant for large \( \text{data}_{\text{ker}} \) and (b) the 3\textsuperscript{rd} term is significant and predominant for large \( \text{data}_{\text{ker}} \) and is insignificant for small \( \text{data}_{\text{ker}} \).
For benchmarks utilizing CUDA streams and overlapping kernel execution with data transfers
the latency is bounded by (2) from above, i.e.:
\[
\delta_{\text{TOT}} \leq 2 \times \left( \delta_{\text{COPY}} + \frac{\delta_{\text{KER}}}{\Theta_{\text{COPY}}} \right) + \delta_{\text{KS}} \quad (7)
\]

This results in a smaller speed-up \( G_{\text{GPU}} \) for such benchmarks. Applying the Amdahl’s law we
get an expression for the total benchmarks speed-up \( G_{\text{TOT}} \):
\[
G_{\text{TOT}} = \frac{1}{\%_{\text{CPU}} + \%_{\text{GPU}} \times G_{\text{GPU}}} \leq G_{\text{GPU}} \quad (8)
\]

The total speed-up is proportional to the percent of the total execution time that was spent
executing the CUDA code and is bounded by the CUDA speed-up \( G_{\text{GPU}} \).

From (5) and (8) we see that the following factors result in higher performance benefits from
fusion:

- High benchmark kernel throughput \( \Theta_{\text{KER}} \)
- Small benchmark input data size \( \text{data}_{\text{TOT}} \)
- Many kernel invocations (large \( n_{\text{KER}} \))
- Long time spent in CUDA code relative to the x86 code (large \( \%_{\text{GPU}} \))

From (5) we also note that for larger problem sizes, the effect of the total latency \( \delta_{\text{TOT}} \) is
amortized by the data size and leads to a reduction of the benefits from fusion.

### 3.2 Performance with Zero-Overhead Coherence

Using the Fused FusionSim simulator version and leveraging the partial separation of functional
from timing simulation in FusionSim we initially disable the software-based coherence
mechanism to measure the maximum attainable performance with no memory coherence
overhead. We do so to obtain an upper limit on the performance benefits that are possible.
Different coherence mechanisms may incur lower overheads compared to our software-enforced coherence mechanism.
For this experiment we assume a fused system with an optimistic 0.1 µsec kernel spawn latency and no memory coherence overhead. **Figure 10** reports relative performance improvement over two discrete baselines. The first (blue bars) uses a 100 µsec kernel spawn latency and the second uses a 10 µsec latency (red bars). We observe that the performance boost due to fusion varies substantially depending on the benchmark, the data input size (compare `gaus_4` and `gaus_128`), and the kernel spawn latency of the discrete system.

**Figure 10.** Speed-up of the Rodinia benchmarks on a fused system with 0.1 µsec kernel spawn latency and no memory coherence overhead.

Focusing on the results with the 100 µsec latency (blue bars) we observe benchmarks that see substantial performance benefit from fusion, such as `gaus_4` and `bfs_small`. Most other benchmarks see modest improvements which can be as high as 4.90x for `pathfinder` and as low as x1.08 for `nn`. When the kernel spawn latency is 10 µsec the benefits of fusion are lower, however they persist for all benchmarks that saw benefits when the kernel spawn latency was higher. For the rest of this study and unless otherwise noted, we will be using a baseline discrete system with a 10 µsec kernel spawn latency.
Results

FIGURE 1 reports the performance of fused system with three different kernel spawn latencies of 0.1, 1, and 10 µsec. Performance is normalized over that of a discrete system with the optimistic 10 µsec latency. Fusion benefits remain mostly unaffected even when kernel spawn latency on the fused system is as high as 1 µsec. When fusion does not reduce the kernel spawn latency (10 µsec), the performance benefits are lower. Even so, we see improvements as low as 9% (pathfinder) and as high as 93% (gaus_4). Unless otherwise noted, the rest of the experiments use the optimistic 0.1 µsec kernel spawn latency for the fused system.
Figure 12. Overhead of the software-based memory coherence mechanism in terms of benchmark execution time.

3.3 Performance with Software Coherence

Figure 12 shows the performance overhead due to the memory coherence mechanism on the fused system. The performance overhead of the coherence mechanism is minor (less than 2%) for most benchmarks, while reaching as high as 5% for bfs_small.
3.4 The Effect of the Input Data Size

We analyze how different sizes of data input affect fusion performance benefits. We focus on two benchmarks that benefit significantly from fusion, *bfs* and *gaus*. **Figure 13** reports the speed-up for the *bfs* benchmark for two input sets. The speed-up of *bfs* reduces as the data input size increases.

**Figure 14** presents speed-ups for the *Gaussian* benchmark for different problem sizes. Focusing on the speed-up of the 10 µsec latency discrete system we note that the benchmark reaches high speed-up of 9.72x with a small problem size, while for a medium problem size the speed-up reduces to 1.84x. The execution of the *Gaussian* benchmark supplied with medium-size input (*gaus_256*) lasts only 10 milliseconds on the discrete system. Therefore, the speed-up due to fusion only becomes significant for small problem sizes of less or close to 10 milliseconds of execution time. This happens since, as the size of the benchmark data input increases, the latency overheads become insignificant compared to the total benchmark execution time and, therefore,
Results

no significant speed-up is attained by elimination or reduction of those latencies. The behaviour is explained and confirmed by the 2\textsuperscript{nd} term of the analytical derivation in (5).

For larger input data sizes, the latency of the CUDA memory copy operations and the kernel spawn latency become insignificant and the benchmark performance on the fused system approaches the performance on the discrete systems.

The fact that smaller input sets get higher speed-ups with fusion will encourage software developers to take a chance porting smaller sections of code to the GPU.

**Figure 14.** Speed-up of *Rodinia Gaussian* benchmark on a fused system for different data sizes. The speed-up is inversely proportional to the benchmark data input size.
3.5 Results Analysis

Figure 11 indicates that gaus and bfs significantly benefit from fusion when executed with small data inputs. Their performance improves by 9.72x and 4.28x relative to the 10 µsec discrete baseline respectively. On the other hand, nn's performance improves only by 1.05x. To explain this behaviour we look at the computation pattern of gaus and bfs and compare it to the computation pattern of nn.

Gaussian exhibits the following computation pattern:

1. Copy input data from the host to the device
2. For multiple iterations do:
   a. Launch kernel #1
   b. Launch kernel #2
3. Copy output data from the device to the host device

BFS exhibits a similar computation pattern:

1. Copy input data from the host to the device
2. For multiple iterations do:
   a) Copy data from the host to the device
   b) Launch kernel #1
   c) Launch kernel #2
   d) Copy input data from the device to the host device
3. Copy output data from the device to the host device

On the other hand, the computation pattern of nn is quite different:

1. Copy input data from the host to the device
2. Launch kernel computation
3. Copy input data from the device to the host device
Considering the computation patterns of bfs and gaus we note that they incur a substantial latency overhead due to multiple kernel spawns and multiple memory transfers. On the other hand nn performs only one kernel spawn and only two memory copies independent of the problem size and, therefore, accumulates only a minor latency overhead.

The differences between bfs and gaus on one hand and nn on the other are also reflected in Table 4. Note that for the same order-of-magnitude input data size the number of kernel spawns of gaus is around 100 times that of nn. Bfs also exhibits one order-of-magnitude greater number of kernel spawns and memory copies.

The analytical model (5) can be used to explain the experimental results. Figure 16 reports

\[
\frac{\Delta_{\text{data}}}{\text{TOTAL}} \text{ from (5) which is the effective computation latency per unit of input data. The total latency } \delta_{\text{tot}} \text{ of a single computation iteration has been calculated as } (10 \times n_{\text{KER}} + n_{\text{MEM \_CPY}}) \times \alpha,
\]

where \( \alpha \) is some constant that gets eliminated in the normalization. The factor 10 accounts for an order-of-magnitude higher value of the kernel spawn latency relative to the latency of CUDA memory copy operations. All measurements are normalized to the value of backprop to make comparisons across benchmarks easier. Backprop experienced the least benefit from fusion as per Figure 10.
Figure 15 shows the kernel processing throughput per benchmarks in Bytes/sec. The model in (5) predicts that fusion speed-up is proportional to the kernel processing throughput. Bfs has the highest throughput among the benchmarks, which explains its high speed-up. For gaus the throughput is medium/low but gets compensated by the extremely high values of the normalized latency in Figure 16 thus resulting in high speed-up for the benchmark. We also note the low value of the throughput for the nn that additionally contributes to the low speed-up. For gaus and nn the value $\frac{\Delta_{TOTAL}}{data_{TOTAL}}$ is the greatest and the smallest among the benchmarks, respectively. This explains the high and the low speed-ups exhibited by the two benchmarks.
Figure 16. Graph of $\frac{\Delta_{TOTAL}}{data_{TOTAL}}$ multiplicand in the speed-up formula (5) normalized to that its value for heartwallbackprop. Note the large value of the expression for gaus_4 and the smallest one for heartwallnn.
Chapter 4. Related Work

Wong and Aamodt conducted a limit study of the potential benefit of the tighter integration of a CPU and a GPU reaching the conclusion that latency and bandwidth of communication link have comparatively minor effects on performance [9].

Hetherington et al. reported up to 7.5X performance increase compared to the CPU when executing the key-value look-up handler on an existing integrated CPU+GPU system [10].

Gregg and Hazelwood analyzed the overheads of moving data between CPU and GPU [11].

Gelado et al. suggested using page permissions to automatically detect when to move data from CPU to GPU [12].

Yang et al. built a simulator similar to the fused version of FusionSim and used to evaluate a CPU-side prefetcher for GPU data into the shared L3 cache [14].

Jablin [15] talks about using the compiler to help automatically moving data between CPU and GPU.
Chapter 5. Future work

This work can be extended in several ways. We study a reasonable fused system configuration. Several other options exist and are worthwhile investigating. For example, future work may look at alternate coherence mechanisms, memory hierarchy organization and policies, physical distribution of CPU and GPU resources. We study existing GPU benchmarks that were developed for discrete systems. They tend to have coarse grain CPU and GPU phases which helps amortizing the high communication costs of discrete systems. Existing applications also exhibit little overlap between the CPU and GPU. Future work may investigate other applications that have finer compute phases and may thus benefit more from fusion and that have higher overlap of execution between the CPU and the GPU. Even with these limitations, we believe that this study is valuable as it demonstrates the tradeoffs and benefits that ought to be expected from fusion for existing applications.

FusionSim provides a simulation environment where many fused architecture schemes can be studied and evaluated. FusionSim can be extended to simulate specific fused architecture designs.

This work provides an experimental study of the performance of fused system for heterogeneous workloads with small or medium benchmark input data sizes. As predicted by the analytical derivation of the speed-up, the 3rd term of (6) might significantly contribute to the speed-up for benchmarks with large input data sizes. Future work may confirm this prediction by simulation.
Chapter 6. Conclusion

We present FusionSim, a modeling framework capable of cycle-accurate simulation of x86-based systems with a) a CPU and a GPU on the same die (fused), and b) a CPU and a GPU connected as separate components (discrete). We use FusionSim to characterize the performance of the Rodinia benchmarks on fused and discrete systems.

We derive an analytical model for the performance benefit from fusion and identify that the following benchmark characteristics directly contribute to the speed-up: a) high benchmark kernel throughput; b) small benchmark input data size or large communication vs. computation ratio; c) multiple kernel invocations and d) large share of the total execution time spent in the CUDA code. We confirm our analytical model experimentally, and take a closer look at the communication and compute structure of some benchmarks. We demonstrate that for benchmarks that benefit a lot from fusion, a 9.72x speed-up is possible for small problem sizes. This speed-up reduces to 1.84x with medium problem sizes.

Using FusionSim we study the dependence of the speed-up on the kernel spawn latency of the fused system and the discrete system. We demonstrate that an order-of-magnitude reduction of the latency on the fused system is required for high speed-up results.

Further, we study a simple, software-managed memory coherence solution for the fused system. We find that it imposes a minor performance overhead of 2% for most benchmarks. We demonstrate that the overhead can be as high as 5% for some benchmarks.
Chapter 7. References or Bibliography


Appendix A: FusionSim Manual

Vitaly Zakharenko <vitaly.zakharenko@utoronto.ca>

Under Supervision of

Prof. Andreas Moshovos, University of Toronto

This document describes Release 2.0.1 of FusionSim.
Last revised on April 24, 2012.

Latest version of this document & latest release are available at

www.fusionsim.ca
Contents

1. Introduction ........................................................................................................................................... 40
   1.1. Introducing FusionSim .................................................................................................................... 40
   1.2. The two simulator flavours ............................................................................................................ 40
   1.3. Fused simulator version .................................................................................................................. 41
   1.4. History ........................................................................................................................................... 42
   1.5. Copyright ....................................................................................................................................... 42

2. Simulator internals ................................................................................................................................. 43
   2.1. Basic Structure .............................................................................................................................. 43
   2.2. Simulator setup and initialization .................................................................................................. 43
   2.3. Simulation ...................................................................................................................................... 44
   2.4. Simulation loop function calls ....................................................................................................... 45
   2.5. Modeled memory system – Fused version specific ...................................................................... 46
       2.5.1. Memory address spaces .......................................................................................................... 46
       2.5.2. Shared virtual memory address space .................................................................................... 46
       2.5.3. Cache coherence ...................................................................................................................... 46
   2.6. Changes to PTLsim ....................................................................................................................... 46
       2.6.1. FusionSim’s memory/cache system ......................................................................................... 47
   2.7. Changes to GPGPU-Sim .................................................................................................................. 48

3. Changes in CUDA API for heterogeneous system – Fused version specific .................................. 49

4. File system structure ............................................................................................................................. 50

5. System requirements ............................................................................................................................. 51

6. FusionSim’s limitations ........................................................................................................................ 51

7. Building FusionSim .............................................................................................................................. 51
   7.1. Resolving GPGPU-Sim dependencies ........................................................................................... 51
   7.2. Resolving PTLsim dependencies .................................................................................................. 52
1. Introduction

1.1. Introducing FusionSim

FusionSim is primarily intended for simulation of execution of general-purpose CUDA workloads on shared memory x86 systems. It simulates both an x86 out-of-order CPU, a CUDA-enabled GPU, CPU/GPU interconnect and a memory system. To the best of our knowledge, it is the first simulator available to the open research community that provides this functionality.

FusionSim simulates execution of standard 32-bit Linux executable files that (optionally) make use of NVidia CUDA API. Benchmark executables are created through standard compilation procedure and are fed to FusionSim without any modifications. All x86 code of the benchmark is simulated on the cycle-accurate out-of-order 32-bit user-space PTLsim CPU simulator. All CUDA API calls (if any) are simulated on GPGPU-Sim GPU simulator. CPU and GPU simulation/execution happens in parallel as it does on an actual machine.

The origin of FusionSim’s name is twofold. FusionSim is a simulator built by fusion of two existing simulators, user-space 32-bit PTLsim CPU simulator and GPGPU-Sim v.3.0.1b GPU simulator. Additionally, we consider FusionSim as a step towards research of fused CPU+GPU designs implementing both units on the same die and partially sharing the memory hierarchy. We hope that future releases of this simulator will implement various models of GPU/CPU interconnect and memory sharing and facilitate research in that area.

1.2. The two simulator flavours

FusionSim comes in two flavours: discrete version and fused version.

- **Discrete FusionSim** versions model a standard discrete CPU/GPU system.
  In a standard discrete system the CPU and the GPU each operate on distinct memory spaces. The memory spaces are distinct both logically (separate memory address spaces) and physically (CPU and GPU each have their private DRAM). In such systems CPU/GPU communication involves copying memory blocks back and forth between CPU DRAM memory and GPU DRAM memory.
via the PCI-E link.

- **Fused FusionSim** versions model a heterogeneous single-chip CPU/GPU system. Some of the recent real-life examples of such systems are AMD’s Lliano and Intel’s Sandy Bridge. In a heterogeneous system both CPU and GPU are implemented on the same chip and share the same DRAM. This implementation alleviates the need to transfer blocks of data between distinct DRAM memories thus reducing bandwidth limitations and latency overheads of PCI-E interconnect.

### 1.3. Fused simulator version

**Fused FusionSim** models x86 out-of-order CPU using cycle-accurate [PTLsim](#) simulator. The GPU is modeled by [GPGPU-Sim](#) simulator. CPU cache hierarchy is modeled by highly-configurable and elaborate [MARSSx86](#) cache system.

The CPU and the CPU cache hierarchy are integrated (as a single block) into GPGPU-Sim in place of one of the GPGPU-Sim's Processing Clusters. Like the Processing Clusters in GPGPU-Sim, the CPU cache hierarchy (L1, L2 and optional L3) is interfacing with GPGPU-Sim's Intersim network-on-chip (NOC) model. The NOC Intersim model (derivative of [BookSim](#) NOC simulator) connects LLC with all Processing Clusters and the CPU cache hierarchy.

The CPU and the GPU share LLC while maintaining private L1 and L2 caches. This results in a cache coherence problem, which is resolved by selective flushing of private caches to LLC/DRAM (see documentation for more details).

The CPU and the GPU share the ‘global’ memory address space (following CUDA terminology) that is mapped to the shared DRAM.

Diagram of the modeled heterogeneous system:
NOTE:

- Currently, just one PTLsim core is supported.
- L3 is optional. By default, the simulator is configured to use L2 as the last level cache within the CPU processing block (enclosed in the blue transparent rectangle).

1.4. History

FusionSim was developed by Vitaly Zakharenko under supervision of Prof. Andreas Moshovos in the University of Toronto.
PTLsim was developed by Matt T. Yourst <yourst@yourst.com>.
GPGPU-Sim was developed by Prof. Tor M. Aamodt, et al., in the University of British Columbia.
MARSS was developed by CAPS Research group in the State University of New York at Binghamton.

1.5. Copyright

Any non-commercial use of FusionSim is allowed under condition that the names of Vitaly Zakharenko and Andreas Moshovos are cited on any publication that uses results obtained through the use of FusionSim or any program produced from any part of FusionSim's source code, which is not entirely a part of PTLsim, GPGPU-Sim or MARSSx86.
2. Simulator internals

2.1. Basic Structure

FusionSim is composed of three main components:

- **PTLSim**
  - modified 32-bit user-space PTLSim (see 2.5)
  - is part of fusionsim executable

- **GPGPU-Sim**
  - modified GPGPU-Sim v. 3.0.1 (see 2.6)
  - is part of fusionsim executable

- **libcudart.so**
  - NVidia’s dynamic library replacement that bears the same so-name and implements the same API
  - is standalone library, which is dynamically linked to the benchmark at runtime

After libcudart.so library has been loaded and just before the 1st instruction at benchmark program entry is simulated the virtual memory space of the simulator process consists of two logical process spaces

a) Simulator process space
  - includes segments of PTLSim and GPGPU-Sim components
  - includes private simulator stack and private simulator heap (separate from benchmark)
  - is executed on the actual host CPU

b) Benchmark process space
  - includes segments of the benchmark and of libcudart.so
  - includes private benchmark stack and private benchmark heap (separate from simulator)
  - is executed on the virtual PTLSim's CPU

Since both the virtual PTLSim CPU and the actual CPU of the simulator host machine are 32-bit x86 and share the same virtual memory space it might be confusing to distinguish x86 the code executed on virtual PTLSim CPU from code, which is executed on the actual CPU. Therefore, I clarify that PTLSim and GPGPU-Sim are executed on the actual CPU, while the benchmark and FusionSim's libcudart.so are executed on the virtual PTLSim CPU.

Since libcudart.so is just a small and almost dummy implementation of the CUDA RT library it only adds a completely insignificant overhead to the benchmark.

2.2. Simulator setup and initialization

On FusionSim executable start-up it performs fork and exec system calls (the latter with benchmark process image) effectively creating a child *benchmark* process. The child process is stopped at this point. Then the parent FusionSim process uses ptrace to inject its process image into the virtual address space of the child benchmark process. At this point the address space of the child process contains both the FusionSim process image and the process image of the benchmark. Execution of the child process is
then restarted (by setting the IP register appropriately) from a specific set-up function originating in FusionSim's process image. This function arranges a separate heap and stack for use by the actual CPU of the host machine.

At this point FusionSim simulator is ready to execute in the child process image using its own stack and heap and not interfering in any way with heap and stack of the simulated benchmark.

The execution of FusionSim in the child process is started from the _start symbol, PTLsim virtual CPU is initialized (by FusionSim) and gets ready to execute the x86 code of the benchmark.

From this point on we will forget about existence of the parent FusionSim process and will refer to the child as *the* FusionSim process.

Execution/simulation of the benchmark starts from the entry point of the interpreter (i.e., linux dynamic loader). Once the interpreter code execution is finished by PTLsim’s virtual CPU the virtual process space contains FusionSim's libcudart.so and other dynamic libraries that the benchmark is linked to. At this point virtual CPU proceeds towards execution/simulation of the benchmark starting from the program entry point (_start symbol by default).

If the benchmark makes use of CUDA API it will call __cudaRegisterFatBinary API function at some point before any other API CUDA call. On this call the virtual CPU of PTLsim will execute __cudaRegisterFatBinary() implementation x86 code of FusionSim's libcudart.so and an instance the virtual GPU of GPGPU-Sim will be created and initialized. (FYI, all API implementations in FusionSim's libcudart.so are very light, their purpose being to provide an indication to FusionSim that the API has been called with such and such arguments. More information on libcudart.so follows below.)

From this point on, both the virtual CPU of PTLsim and the virtual GPU of GPGPU-Sim are instantiated, initialized and executing "in parallel" (which does not imply multithreading).

### 2.3. Simulation

FusionSim defines a virtual common clock, from which both the CPU and the GPU core clocks are derived through frequency multipliers (values of both the common clock frequency and the multipliers are set in configuration files). FusionSim simulation runs in an (almost) infinite loop, each iteration of which corresponds to one cycle of the common clock.

In each iteration of the loop the virtual PTLsim CPU executes number of CPU cycles equal to that of the CPU frequency multiplier. Similarly, GPU core (and other GPU domains and also the GPU/CPU interconnect) are advanced by time period equal to \( \text{GPU core period} \times \text{GPU frequency multiplier} \). (see ooocore.cpp file).

Following is intended to clarify the functionality of FusionSim's libcudart.so. Be aware that, although FusionSim (i.e. PTLsim & GPGPU) and the benchmark share the same process virtual memory space, the benchmark the benchmark is not allowed to access mapped memory regions belonging to FusionSim (and unmapped regions too). That is because PTLsim maintains records of memory mappings that the benchmark is allowed to access and will gracefully abort the simulation if the benchmark misbehaves. Now, in addition to memory mappings that the benchmark is naturally allowed to access (the ones
acquired with exec and mmap system calls) FusionSim specifies an additional small memory region as accessible to the benchmark. This memory region is used as a kind of gateway for communication between the benchmark and FusionSim.

On each CUDA API call performed by the benchmark the virtual PTLsim CPU executes a dummy implementation of the API in libcudart.so. The API implementation in libcudart.so just stores API call arguments in a structure placed within the call gateway memory region, decrements a non-blocking semaphore and enters a spin lock waiting until the semaphore is incremented by FusionSim. On each GPU core clock cycle FusionSim examines the semaphore, processes the API call and stores the return value (e.g., last CUDA error) in a structure within the gateway memory region. Unless the API call is a synchronous call (e.g., CUDA stream operation in stream zero or synchronous memcpy) the whole processing completes within the same GPU cycle (either because the processing is indeed complete or because the required action is asynchronous and has been queued for further execution). In case of a synchronous CUDA call the semaphore will be posted on the GPU cycle, when the synchronous operation is finished. Whenever the semaphore is posted the virtual CPU of PTLsim leaves the semaphore polling loop, reads the return value and proceeds.

### 2.4. Simulation loop function calls

With every CPU cycle the virtual CPU of PTLsim crunches through the code of the benchmark and, whenever there’s a call to CUDA API, the virtual CPU executes a dummy CUDA API implementation of libcudart.so (i.e., of cuda_runtime_api_if.cc). This dummy implementation of a CUDA API stores API call arguments in cuda_runtime_api_if structure at a fixed address, calls wait on a semaphore in the same structure and enters a spin lock polling the semaphore until it gets unlocked. (Please notice that a non-blocking implementation of semaphore wait is used so the execution thread is never suspended). On next gpu_cycle() call from infinite simulation loop in OutOfOrderMachine::run() the function calls ServeCudaApiCall() implemented in cuda_runtime_api.cc. ServeCudaApiCall() examines the semaphore and detects the CUDA API call pending processing by the GPU. The call is directed to the specific processing function defined in cuda_runtime_api.cc. The API processing function executes some logic, optionally updates the stream_manager class instance with a new task/event, sets return value in cuda_runtime_api_if and returns. Next, gpu_cycle() calls StepOneGpuCycle() method of StreamOpConsumer class (which is the top-level GPU abstraction). StepOneGpuCycle() method advances GPU by time equal to one GPU core cycle period and returns a boolean value. This value indicates whether the CPU should be blocked according to the records of stream_manager or it is free to continue. The CPU will be blocked if the CUDA API call was a synchronous operation (e.g., stream operation to stream zero or synchronous memcpy). The CPU will be free to proceed if the CUDA API call is asynchronous (e.g., asynchronous memcpy) or one that takes zero time to execute (e.g., GetDeviceProperties CUDA call). Depending on the return value of StepOneGpuCycle() the semaphore will be posted or not posted (if CPU is blocked). If the CPU is blocked it will be unblocked on one of the subsequent gpu_cycle() calls when one or more operations are finished and stream_manager indicates the state of the CPU as not blocked.

In order to improve simulation performance gpu_cycle() function returns a boolean value indicating whether the CPU is blocked. If blocked, the cycles of CPU are voided resulting in faster simulation. Please note that statistical records will correctly indicate the number of cycles that the CPU was executing.
2.5. Modeled memory system – Fused version specific

2.5.1. Memory address spaces
The CPU and the GPU share the ‘global’ address space that is mapped to the shared DRAM.

CPU and GPU share LLC and DRAM. Physical address space is shared and is mapped to the DRAM.

2.5.2. Shared virtual memory address space
The simulator models/assumes a system that implements virtual memory addressing for the GPU so that the virtual to physical address translation is the same for both the GPU and the CPU (i.e., same MMU). However, the simulator does not model the actual virtual to physical translation neither on the CPU nor on the GPU side and just uses virtual addresses as the physical ones. This is an allowable simplification assuming the translation is outside of the critical path and has no considerable impact on performance.

2.5.3. Cache coherence
CPU cache hierarchy (L1, L2 and the optional L3) is private to the CPU cores and is not accessible from the GPU. The same is true with respect to accessibility of GPU’s L1 cache by the CPU.

Since the GPU and the CPU share physical address space it is possible that two cache blocks associated with the same physical address are present (and indicated as valid) and hold different data. For instance, two distinct data lines associated with the same physical address may be present simultaneously in L1 data cache of a CPU core and in L1 data cache of a GPU core. There is a potential for cache incoherence.

The simulator resolves this problem by selective flushing of modified cache lines to the shared LLC/DRAM prior to kernel invocation. I.e., if a CUDA kernel operates on data block extending from the base address of 0x8000 up to 0xF000 then, just before the CUDA kernel is launched, all modified blocks in the CPU cache hierarchy in the address range (0x8000; 0xF000) are flushed to the shared LLC/DRAM.

Similarly, on each CUDA kernel completion the L1 caches of GPU cores are flushed to the shared LLC/DRAM.

This coherence methodology requires from the CUDA programmer to specify memory regions operated/accessed by a kernel prior to each kernel invocation. Please see the section ‘Changes to the CUDA API’ for details on the additional API calls required by the coherence mechanism.

2.6. Changes to PTLsim
As you might know, user-space PTLsim is a simplified version of a full-system PTLsim simulator. Both PTLsim versions were supposed to be built from the same source code by using different pre-processor definitions and flags. Therefore, the user-space PTLsim inherited some features from its full-system counterpart. These features and the requirement to switch b/w simulated and native execution partly over-complicated the code base of the user-space version.

Thus, the original user-space version does not use standard C/C++ library implementing all necessary libc functionality in its own sources. Also, original PTLsim's building procedure is very ad-hoc (e.g., ad-hoc
linkage scripts, no use of g++ compilation driver, ad-hoc execution of static c'tors/d'tors, lots of flags etc.). This resulted in PTLsim being not-extensible and failing to compile on any g++ version except the one it was developed for.

I have considerably modified PTLsim (the part responsible for simulator set-up) so that it can be compiled using a simple more-or-less standard make file following standard compilation procedure and using libc. PTLsim's "superstl" and multiple other files have been rewritten or modified to enable extensibility of the code base. Structure of the PTLsim project has been changed. To achieve extensibility and standard compilation procedure I had to disable the very useful feature of PTLsim that allows switching back-and-forth between native and simulated execution (please keep this in mind while reading PTLsim's manual. I also removed configuration options pertaining to this functionality).

PTLsim's simulation engine (both sequential and out-of-order) has not been modified significantly and has retained all the features, including precision and high performance of the original PTLsim.

The original PTLsim cache system is slightly simplistic, insufficiently configurable and models only write through caches with unified L1 data/instruction cache. In version 1.1 of FusionSim, in order to improve configurability and precision of memory/cache system modeling I replaced the original cache/memory system with that of MARSSx86 simulator (also a PTLsim derivative). The new cache/memory system of FusionSim is described below.

Please note, that FusionSim v.1.0 uses original PTLsim cache system for memory access modeling.

**2.6.1. FusionSim's memory/cache system**

In current revision of FusionSim (rev. 1.1) the original cache hierarchy of PTLsim (modeling of caches, memory and interconnect in-between) has been replaced with the cache/memory hierarchy of MARSSx86 simulator. The latter features higher configurability, modularity and precision offering modular configurable units of regular/coherent caches, memory, point-to-point interconnect, bus interconnect, etc.

Current revision of FusionSim takes advantage of these features of MARSSx86.

FusionSim models one out-of-order CPU core connected to separate data and instruction L1 caches. L1 caches are both connected to the same L2 cache. The latter is further connected to an L3 cache. Each of the caches above is represented by a separate instance of CacheController class and is configurable with respect to the replacement policy (write-through or write back) and various other parameters. L3 cache is connected to a unit modeling banked DRAM memory.

Following MARSSx86 files have been added to FusionSim:

- **cacheConstants.h**
  - Defines constants that determine configuration of cache/memory units and the CPU memory controller.

- **cacheController.(h/cpp)**
  - Declare & define class modeling a non-coherent cache.

- **cacheLines.h**
Appendix A

- **controller.h**
- **cpuController.(h/cpp)**
  - Declare & define class modeling the CPU memory controller.
- **interconnect.h**
- **memoryController.(h/cpp)**
  - Declare & define class modeling the DRAM.
- **memoryHierarchy.(h/cpp)**
  - Declare & define the topmost abstraction of the memory hierarchy.
- **memoryRequest.(h/cpp)**
- **p2p.(h/cpp)**

Parameters of all units belonging to the cache hierarchy can be configured by modifying corresponding constants in ‘cacheConstants.h’ and recompiling. For more extensive modifications (e.g., replacing non-coherent cache units with coherent ones, changing cache interconnect) please consider the constructor of MemoryHierarchy class in ‘memoryHierarchy.cpp’ file.

### 2.7. Changes to GPGPU-Sim

As you probably know, GPGPU-Sim replaces NVidia’s libcuda.so with its own library bearing the same name and implementing the same CUDA API. If LD_LIBRARY_PATH environment variable is set to point to the location of GPGPU-Sim's dynamic library then, on benchmark execution, GPGPU-Sim's library is loaded in place of NVidia’s one and simulates GPU processing of CUDA API calls.

FusionSim follows a similar methodology replacing NVidia’s dynamic library. However, FusionSim's dynamic library is almost a dummy: all it does is just storing CUDA API calls in a specific structure at a fixed address within the process's virtual address space. Please see "./gpgpusim/libcuda/cuda_runtime_api_if.cc", which is the only source of FusionSim's libcudart.so.

In GPGPU-Sim most of the files in folders "/gpgpusim/libcuda" and "/gpgpusim/src" have been almost entirely rewritten. You should especially consider the following files:

- "./gpgpusim/libcuda/Makefile" has been considerably modified
  
  The make file now only uses "cuda_runtime_api_if.cc" (that includes "cuda_runtime_api_if.h") to build libcudart.so dynamic library. The file "./gpgpusim/libcuda/cuda_runtime_api_if.cc" is now compiled to an object file to be later on linked to FusionSim.

- "./gpgpusim/libcuda/cuda_runtime_api_if.h" is a new file
  
  This file defines a structure used as a gateway for passing CUDA API calls from the benchmark to FusionSim. It is included both into "cuda_runtime_api.cc" (that becomes a part of FusionSim) as
well as into "cuda_runtime_api_if.cc" (that becomes a part of the benchmark). The structure declared in this file is allocated by FusionSim and placed at a fixed virtual memory address, is known and accessible to both benchmark and FusionSim and used as an API call gateway.

- "./gpgpusim/libcuda/cuda_runtime_api.cc" has been considerably modified

This file is not a part of libcudart.so any more, but a part of FusionSim. It contains actual (not dummy like in cuda_runtime_api_if.cc) implementations of CUDA API functions.

- "./gpgpusim/src/gpgpu_if.h" is a new file

This file declares just a single function (bool gpu_cycle()) that is implemented in "./gpgpusim/src/gpgpusim_entrypoint.cc". This function processes pending CUDA API calls (if any) by calling ServeCudaApiCall() function implemented in "cuda_runtime_api.cc". Additionally, the function advances GPU time by one GPU core cycle by calling StepOneGpuCycle() method of class StreamOpConsumer. Multiple GPGPU-Sim's domains may get simulated (have their clock cycle advanced) in course of this function call.

- "./gpgpusim/src/gpgpu_sim_entrypoint.cc" has been entirely rewritten

It implements gpu_cycle() function and other functions called on creation/initialization/destruction of GPGPU-Sim's instance and instances of stream_manager and StreamOpConsumer classes.

- "./gpgpusim/src/stream_manager.cc" has been entirely rewritten

stream_manager class manages all stream operations and events active/scheduled or registered in GPGPU-Sim. One of the main differences between this implementation and the original one is that all methods of stream_manager are non-blocking.

- "./gpgpusim/src/StreamOpConsumer.(h/cpp)" is a new file

This class uses stream_manager and gpgpu_sim classes and implements the top-level GPU abstraction.

### 3. Changes in CUDA API for heterogeneous system – Fused version specific

Since the simulator models a heterogeneous system, where both physical and virtual address spaces are shared between the CPU and the GPU, there is no more need for copying data blocks between the distinct memory spaces (like in a standard discrete system).

Thus, the CUDA API calls such as ‘cudaMemCopy’ (whether from device to host or in opposite direction) are not needed any more. Similarly, there is no more need for CUDA’ malloc’ calls allocating global memory on the device.

On kernel invocation we supply pointers (as kernel arguments) to the shared virtual memory space and don’t need to care about separate pointers for device global memory and host memory any more.
However, the heterogeneous system does not only cut on number of CUDA APIs that we need, but also requires some minor extension of CUDA API. Thus, in order to maintain cache coherence between CPU and GPU cache hierarchies the programmer needs to specify ranges of memory addresses accessed by a kernel prior to kernel invocation (see ‘Coherence’ section for more details).

In order to extend the CUDA API (and simulate write-back memory accesses caused by flushing) you will need to link your CUDA executable to ‘libcuda_ext.so’, that becomes available in ‘fusionsim/gpgpusim/lib’ folder once you successfully make FusionSim. This library implements just a single additional CUDA API ‘cudaSelectivelyFlush(void* baseAddr, int size)’ that should be called prior to every kernel invocation to generate (and simulate) a series of cache line flush requests. These requests will be issued just before kernel invocation and ensure that all data accessed by the GPU is up-to-date in the shared LLC/DRAM.

For precise prototypes of the extension functions please address/include the file ‘fusionsim/gpgpusim/libcuda/cuda_runtime_api_ext.h’.

Note: If you don’t link your executable to ‘libcuda_ext.so’, the simulation will still work fine (because functional and timing simulations are implemented on separate ’simulation levels’). However, the flush requests will not be generated and will not be simulated.

4. File system structure

Following folders/files may be of interest:

- "." (FusionSim’s home folder)
  This folder contains sources of the modified PTLSim, which is the main constituent part of FusionSim.

- "/configs"
  This folder contains all default configuration files of FusionSim. See section (7.3) for description of configuration files.

- "/dist"
  This folder is created on "make pack" command and contains compressed tar archives of project source code and documentation.

- "/gpgpusim"
  This folder contains all GPGPU-Sim's source code. Structure of GPGPU-Sim's sources has been preserved to simplify integration of FusionSim with future releases of GPGPU-Sim.

- "/gpgpusim/setup_env"
  This file sets up environment variables that are used for both building GPGPU-Sim components of FusionSim and running simulations. Therefore, this file is included into both "/gpgpusim/Makefile" and "/sim/simulate.bash".
5. System requirements

You will need a 32-bit Linux. The simulator will not compile on 64-bit OS.

FYI, the simulator has been developed on 32 bit Ubuntu 10.04 LTS (Lucid Lynx) for CUDA Toolkit 3.1.

6. FusionSim's limitations

- FusionSim only simulates 32-bit executable files and only compiles on 32-bit Linux. This restriction is just because this is the 1st release of FusionSim. There is no considerable difficulty in extending FusionSim so that it runs on 64-bit machines and simulates 64-bit executables since both PTLsim and GPGPU-Sim support 64-bits word size.

- FusionSim only simulates CUDA (or standard non-GPU executables) and will not simulate OpenCL. Reason is the same as in (1).

- FusionSim only supports CUDA API as defined in version 3.1 of "cuda_runtime_api.h" of NVidia. You might be aware that GPGPU-Sim does not implement some of the API functions in "cuda_runtime_api.h". FusionSim will abort simulation if it detects an API call to an unimplemented CUDA function.

- FYI, FusionSim is based on ver.3.0.1 of GPGPU-Sim and (2009-03-14 rev 229) release of PTLSim.

7. Building FusionSim

7.1. Resolving GPGPU-Sim dependencies

Since FusionSim uses GPGPU-Sim v.3.0.1 as one of its components you need to ensure that all tools & libraries required by GPGPU-Sim v.3 are present on your system. Please refer to "./gpgpusim/README" for more information. It would be a good practice to ensure GPGPU-Sim v.3 works on your system prior to installing FusionSim.

Please specify the actual paths by setting the following environment variables to their correct values in "./gpgpusim/setup_env":

- CUDA_INSTALL_PATH
- CUDAHOME
- NVIDIA_COMPUTE_SDK_LOCATION
7.2. Resolving PTLsim dependencies

PTLsim will require libGL.a, libm.a, libz.a (all of them static). Please specify location of the OpenGL library by setting the OPEN_GL_LIB_PATH variable in "/Makefile". Additionally, FusionSim will require boost development headers be installed on your system.

7.3. Building FusionSim

Please set the DEBUG environment variable to 1 if you would like to build a debug version of FusionSim. Run make.

8. Using FusionSim

8.1. Running FusionSim

The simplest way to run a FusionSim simulation is by executing "./.sim/simulate.bash" script in your bash shell (it does not matter from which directory you are running it).

Type following in bash shell: "./simulate.bash", assuming your current working directory is "./.sim".

This script does the following:
- sets all required environment variables
- determines FusionSim configuration (either release or debug) to be used in simulation
- determines the benchmark to be simulated
- optionally feeds non-default configurations to FusionSim (more on it below)
- optionally redirects FusionSim output to non-default output files (more on it below)
- actually starts the simulation

Please read on and examine the script for more information.

8.2. Input/output management

FusionSim uses 3 configuration input files and outputs results to 4 output files.

Each of the input files and output files is opened on a specific file descriptor (e.g., PTLsim's configuration file is always opened on file descriptor #3, while GPGPU-Sim's output file is always opened on file descriptor #9). You will see below how this property can become very useful.

FusionSim uses 3 input files for configuration of PTLsim, GPGPU-Sim and GPGPU-Sim's internal interconnect. A default path is associated with each of the configuration files (they are all in configs directory). There are also hard-coded setting for each configuration.

For each configuration FusionSim will detect whether a non-default file has been redirected to the corresponding file descriptor. If redirection is detected the non-default file redirected to the file descriptor will be used. Otherwise, FusionSim searches for a default file for this configuration. If found, FusionSim
reads and parses setting from that default file. Otherwise, FusionSim outputs a warning that hard-coded settings will be used to the standard output and proceeds.

(NOTE: all examples below assume having properly set the environment variables similar to how they are set in "./sim/simulate.bash" script.)

E.g., if FusionSim is executed using the following bash command "./fusionsim my_benchmark" it will search for default configuration files for each of the configurations (for PTLsim, GPGPU-Sim and GPGPU-Sim's internal interconnect).

However, if FusionSim is executed using the following bash command "./fusionsim my_benchmark 7<~/gpgpusim.config" then FusionSim will read the GPGPU-Sim configuration from file "~/gpgpusim.config" in place of the default file, while still using default files for configuring GPGPU-Sim's internal interconnect and PTLsim.

FusionSim writes to 4 output files, which are PTLsim's output, PTLsim's log file, PTLsim's binary statistics file and GPGPU-Sim's output. A default path is associated with each one of the output files (they are all in your working directory).

For each output file FusionSim will detect whether a non-default file has been redirected to the corresponding file descriptor. If redirection is detected the non-default file directed to the file descriptor will be used. Otherwise, FusionSim will try opening the default file for writing. If failed, FusionSim will write the corresponding output to standard output.

E.g., if FusionSim is executed using the following bash command "./fusionsim my_benchmark" then for each output if will try opening the default output file for writing. If failed then the output will be written to FusionSim's standard output.

However, if FusionSim is executed using the following bash command "./fusionsim my_benchmark 4>~/ptlsim.log" it will try opening the file in path "~/ptlsim.log" for log output. Only if the operation fails will FusionSim consider the default log output file.

Similarly, you can redirect PTLsim's log to standard output by executing FusionSim with the following command "./fusionsim my_benchmark 4>1".

As pointed out in section (7.1), FusionSim is released with "./sim/simulate.bash" script that determines the sources and destinations of all inputs and outputs of the simulation.

Please note, that FusionSim's banner always shows which configuration files were used for configuring which component and which output files were used for which output.

Please also note, that, by default, you will only see the output from your benchmark and critical messages of FusionSim printed to the standard output. However, by modifying "./sim/simulate.bash" script any output can be redirected to stdout/stderr.
8.3. **Configuration files**

FusionSim requires following configuration files for execution:

1. **PTLsim configuration file**

   Path to default configuration file is "./../configs/ptlsim.config" relative to location of the FusionSim executable.

   This configuration file defines properties of the virtual CPU of PTLsim.

   Format of this file is equivalent to that used in original PTLsim. However, some of the options present in PTLsim are not relevant to the modified PTLsim version used in FusionSim and have been removed.

   For a list of all available options and their description please run FusionSim with no benchmark argument and review the output of PTLsim in "ptlsim.out". Alternatively, you can find a list of available options (without description) and their actual settings for a specific simulation in "ptlsim.log" that (by default) should be produced in your working directory on each simulation run.

   In addition to options that were present in PTLsim and were retained following new options have been added:

   - **common-clk-freq**
     
     The value of this option defines the frequency of the virtual common CPU/GPU clock, from which the GPU/CPU clocks are derived.
     
     As described in section (2), all clocks in FusionSim are derived from a virtual common clock by multiplying the common clock frequency by some multiplier.
   - **cpu-freq-mult**
     
     This option determines how many PTLsim CPU cycles occur within a single virtual common clock cycle.
   - **gpu-freq-mult**
     
     This option determines how many GPGPU-Sim core cycles occur within a single virtual common clock cycle.

   To learn more about the format of the configuration file please refer to original documentation of PTLsim in file “./doc/PTLsimManual.pdf”.

   To learn which PTLsim configuration options were actually used in a simulation please consider the PTLsim log file.

2. **GPGPU-Sim's configuration file**

   Path to default configuration file is "./../configs/gpgpusim.config" relative to location of the FusionSim executable.

   This configuration file defines properties of the virtual GPU of GPGPU-Sim. All GPGPU-Sim’s configuration options have been preserved except the following two options:
The option "-inter_config_file" has been removed from GPGPU-Sim.
Format of the option "-gpgpu_clock_domains" now specifies frequencies of domains normalized to the GPU core frequency. Since this normalization would result in 1st parameter being always equal to 1, it is omitted.

Following options were added to GPGPU-Sim configuration file to configure simple latency/bandwidth model of memory-related operations:

- "-lat_memcpy_d2d": Latency of memcpy device-to-device operations (sec)
  Default value is 0.
- "-lat_memcpy_d2h": Latency of memcpy device-to-host operations (sec)
  Default value is 0.
- "-lat_memcpy_h2d": Latency of memcpy host-to-device operations (sec)
  Default value is 0.
- "-lat_memcpy_to_symbol": Latency of memcpy to symbol operations (sec)
  Default value is 0.
- "-lat_memcpy_from_symbol": Latency of memcpy from symbol operations (sec)
  Default value is 0.
- "-lat_memset": Latency of memset operations (sec)
  Default value is 0.
- "-bw_memcpy_d2d": Bandwidth of memcpy device-to-device operations (bytes/sec)
  Default value is 1e+09.
- "-bw_memcpy_d2h": Bandwidth of memcpy device-to-host operations (bytes/sec)
  Default value is 1e+09.
- "-bw_memcpy_h2d": Bandwidth of memcpy host-to-device operations (bytes/sec)
  Default value is 1e+09.
- "-bw_memcpy_to_symbol": Bandwidth of memcpy to symbol operations (bytes/sec)
  Default value is 1e+09.
- "-bw_memcpy_from_symbol": Bandwidth of memcpy from symbol operations (bytes/sec)
  Default value is 1e+09.
- "-bw_memset": Bandwidth of memset operations (bytes/sec)
  Default value is 1e+09.

Settings of the default configuration file are copied from GPGPU-Sim v.3 distribution and simulate NVidia Quadro FX5800.

For more information on GPGPU-Sim options please refer to GPGPU-Sim's documentation.

3. GPGPU-Sim's internal interconnect configuration file

Path to default configuration file is "./.configs/ggpgpusim_icnt.config" relative to location of the FusionSim executable.
This configuration file defines properties of internal interconnect of GPGPU-Sim, which is implemented using BookSim simulator of William Dally, et al. Both format of the configuration file and all its available options are preserved.

Settings of the default configuration file are copied from GPGPU-Sim v.3 distribution and simulate NVidia Quadro FX5800.

8.4. Output files

By default, only benchmark output and critical FusionSim messages are written to standard output. All other outputs are written to 4 different files. However, by redirecting file descriptors from within "./sim/simulate.bash" script you can direct different output to the same file of your choice or stdin/stderr.

FusionSim outputs to following files:

1. PTLsim output file
   Default path is to file "ptlsim.out" in your current working directory.

   This file contains all non-critical output of PTLsim that, in the original version of PTLsim, was sent to the standard output.

2. PTLsim log file
   Default path is to file "ptlsim.log" in your current working directory.

   This is the log file of PTLsim. Depending on the logging level set in PTLsim's configuration file and duration of the simulation this file may become large.

3. PTLsim binary statistics file
   Default path is to file "ptlsim.stats" in your current working directory.

   This is a binary file designed for space and time-efficient collection of statistics in course of PTLsim execution. In order to parse the file, analyze the results and present them in a human-readable format you will need the "ptlstats" executable, which is located in the same folder as "fusionsim" executable. For more information on ptlstats please read the PTLsim manual in "./doc" folder.

4. GPGPU output file
   Default path is to file "gpgpusim.out" in your current working directory.

   This file contains all non-critical output of GPGPU-Sim that, in the original version of GPGPU-Sim, was written to the standard output.
9. Debugging FusionSim

Please use "debug" configuration of FusionSim for debugging. In order to build FusionSim in "debug" configuration please set DEBUG environment variable (by uncommenting the appropriate line within the make file). You will also have to change the configuration to be used for simulation to "debug" by uncommenting the appropriate line within "./sim/simulate.bash".

As described in section (2), by executing "./sim/simulate.bash" script you start the parent FusionSim process that creates a child from benchmark's executable image and injects itself into the child's virtual memory space. Since it is the child FusionSim process that actually performs the simulation it is the child that you will probably want to debug.

Please run simulation by executing "./sim/simulate.bash" script. You will receive two messages from FusionSim asking you to attach debugger to process ID. The specified process ID is that of the child FusionSim process.

Please note that the 1st notice is issued by child FusionSim process just before the virtual CPU of PTLsim starts to execute the code of the dynamic loader. GDB modifies the code of the process being debugged by inserting a break point (0xCC code) into dynamic loader's x86 code. Therefore, attaching GDB on the 1st notice is likely to create a problem: PTLsim is unaware that x86 code has been modified and will try to decode and execute the modified x86 instructions. This will cause PTLsim to crash unless you specifically capture the address, where the break point has been inserted by GDB and modify it to the original x86 code prior to decoding & execution. Therefore, please do not attach GDB on 1st attach message unless you want to debug early stages of PTLsim execution.

The 2nd message is issued when the virtual CPU of PTLSim commits the first instruction on program entry symbol. Please attach debugger after this notice. FusionSim should not crash because of GDB breakpoints in this case.

Wherever you choose to attach the debugger, you will need to load debugging symbols by issuing following command to GDB: "symbol-file ~/fusionsim/bin/debug/fusionsim", assuming "fusionsim" directory is in your home directory.
Appendix B

This appendix provides an example of modifications to the benchmark source code required for execution on the fused heterogeneous system.
1.1. **Original vector addition CUDA SDK example source code**

```c
/* Vector addition: C = A + B.
 * This sample is a very basic sample that implements element by element
 * vector addition. It is the same as the sample illustrating Chapter 3
 * of the programming guide with some additions like error checking.
 */

// Includes
#include <stdio.h>
#include <cutil_inline.h>

// Variables
float* h_A;
float* h_B;
float* h_C;
float* d_A;
float* d_B;
float* d_C;
bool noprompt = false;

// Functions
void Cleanup(void);
void RandomInit(float*, int);
void ParseArguments(int, char**);

// Device code
__global__ void VecAdd(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main(int argc, char** argv)
{
    printf("Vector addition_new\n");
    int N = 50;
    size_t size = N * sizeof(float);
    ParseArguments(argc, argv);
}
// Allocate input vectors h_A and h_B in host memory
h_A = (float*)malloc(size);
if (h_A == 0) Cleanup();

h_B = (float*)malloc(size);
if (h_B == 0) Cleanup();

h_C = (float*)malloc(size);
if (h_C == 0) Cleanup();

// Initialize input vectors
RandomInit(h_A, N);
RandomInit(h_B, N);

// Allocate vectors in device memory
cutilSafeCall( cudaMalloc((void**)&d_A, size) );
cutilSafeCall( cudaMalloc((void**)&d_B, size) );
cutilSafeCall( cudaMalloc((void**)&d_C, size) );

// Copy vectors from host memory to device memory
cutilSafeCall( cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice) );
cutilSafeCall( cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice) );

// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
cutilCheckMsg("kernel launch failure");
#endif

cutilSafeCall( cudaThreadSynchronize() );
#endif

// Copy result from device memory to host memory
// h_C contains the result in host memory
cutilSafeCall( cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost) );

// Verify result
int i;
for (i = 0; i < N; ++i) {
    float sum = h_A[i] + h_B[i];
    if (fabs(h_C[i] - sum) > 1e-5)
        break;
}
printf("%s \n", (i == N) ? "PASSED" : "FAILED");

Cleanup();
}
void Cleanup(void)
{
    // Free device memory
    if (d_A)
        cudaFree(d_A);
    if (d_B)
        cudaFree(d_B);
    if (d_C)
        cudaFree(d_C);

    // Free host memory
    if (h_A)
        free(h_A);
    if (h_B)
        free(h_B);
    if (h_C)
        free(h_C);

cutilSafeCall( cudaThreadExit() );

    if (!noprompt) {
        printf("\nPress ENTER to exit...\n");
        fflush( stdout );
        fflush( stderr );
        getchar();
    }

    exit(0);
}

// Allocates an array with random float entries.
void RandomInit(float* data, int n)
{
    for (int i = 0; i < n; ++i)
        data[i] = rand() / (float)RAND_MAX;
}

// Parse program arguments
void ParseArguments(int argc, char** argv)
{
    for (int i = 0; i < argc; ++i)
        if (strcmp(argv[i], "--noprompt") == 0 ||
            strcmp(argv[i], "-noprompt") == 0)
            noprompt = true;
    break;
}
Appendix B
1.2. Vector addition CUDA SDK example modified for the fused system

/* Vector addition: C = A + B. 
 * 
 * This sample is a very basic sample that implements element by element 
 * vector addition. It is the same as the sample illustrating Chapter 3 
 * of the programming guide with some additions like error checking. 
 */

// Includes
#include <stdio.h>
#include <cutil_inline.h>

// Variables
float* h_A;
float* h_B;
float* h_C;
//float* d_A;
//float* d_B;
//float* d_C;
bool noprompt = false;

// Functions
void Cleanup(void);
void RandomInit(float*, int);
void ParseArguments(int, char**);

extern "C" void cudaSelectivelyFlush(void* addr, int size);

// Device code
__global__ void VecAdd(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main(int argc, char** argv)
{
    printf("Vector addition_new2\n");
    int N = 50;
    size_t size = N * sizeof(float);
ParseArguments(argc, argv);

// Allocate input vectors h_A and h_B in host memory
h_A = (float*)malloc(size);
if (h_A == 0) Cleanup();

h_B = (float*)malloc(size);
if (h_B == 0) Cleanup();

h_C = (float*)malloc(size);
if (h_C == 0) Cleanup();

// Initialize input vectors
RandomInit(h_A, N);
RandomInit(h_B, N);

// Allocate vectors in device memory

// cudaMalloc((void**)&d_A, size) ;
// cudaMalloc((void**)&d_B, size) ;
// cudaMalloc((void**)&d_C, size) ;

// Copy vectors from host memory to device memory
// cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice) ;
// cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice) ;

// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
cudaSelectivelyFlush(h_A, size);
cudaSelectivelyFlush(h_B, size);
cudaSelectivelyFlush(h_C, size);
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(h_A, h_B, h_C, N);
cutilCheckMsg("kernel launch failure");

#ifdef _DEBUG
    cutilSafeCall( cudaThreadSynchronize() );
#endif

// Copy result from device memory to host memory
// cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost) ;

// Verify result
int i;
for (i = 0; i < N; ++i) {
    float sum = h_A[i] + h_B[i];
    if (fabs(h_C[i] - sum) > 1e-5)
        break;
printf("%s \n", (i == N) ? "PASSED" : "FAILED");

Cleanup();
}

void Cleanup(void) {
    // Free device memory
    //if (d_A)
    //    cudaFree(d_A);
    //if (d_B)
    //    cudaFree(d_B);
    //if (d_C)
    //    cudaFree(d_C);

    // Free host memory
    if (h_A)
        free(h_A);
    if (h_B)
        free(h_B);
    if (h_C)
        free(h_C);

cutilSafeCall( cudaThreadExit() );

    if (!noprompt) {
        printf("\nPress ENTER to exit...\n");
        fflush( stdout);
        fflush( stderr);
        getchar();
    }

    exit(0);
}

// Allocates an array with random float entries.
void RandomInit(float* data, int n) {
    for (int i = 0; i < n; ++i)
        data[i] = rand() / (float)RAND_MAX;
}

// Parse program arguments
void ParseArguments(int argc, char** argv) {

for (int i = 0; i < argc; ++i)
    if (strcmp(argv[i], "--noprompt") == 0 ||
        strcmp(argv[i], "-noprompt") == 0)
    {
        noprompt = true;
        break;
    }