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

Vitaly Zakharenko† Tor Aamodt‡ Andreas Moshovos†

†Electrical and Computer Engineering
University of Toronto

‡Electrical and Computer Engineering
University of British Columbia

***Abstract*— 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 or large problem sizes. We study a simple, software-managed coherence solution for the fused system. We find that it imposes a minor performance overhead of 2% for most benchmarks and as high as 5% for some. Finally, we develop an analytical model for the performance benefit that is to be expected from fusion for applications with a simple communication and computation pattern and show that FusionSim follows the predicted performance trend.**

Keywords—CPU and GPU Fusion;

I. Introduction

Heterogeneous systems, comprising a CPU (central processing unit) and a GPU (graphics processing unit), that are both capable of general-purpose computations have started to emerge. A matching heterogeneous workload could take advantage of both the high computing power of GPUs for SIMD-friendly multithreaded computation and the high single-thread performance of CPUs.

Originally, heterogeneous computer systems were discrete where the CPU and the GPU were on separate dies each with its own private DRAM memory. Communication between the CPU and GPU were to be orchestrated by the application by copying data between the CPU and the GPU over a high latency and bandwidth link (PCIe). Fused heterogeneous architectures are now emerging where the CPU and the GPU are on the same die and share the same DRAM memory. Fused systems promise higher performance gains than discrete systems as they obviate the need to explicitly copy data back and forth over a long latency link.

We use FusionSim [2] to demonstrate the potential benefits of fused systems. FusionSim is an open-source modeling framework capable of cycle-accurate simulation of a complete x86-based computer system with a CPU and a GPU. We compare a discrete system that is representative of a commercially available solution and a hypothetical, yet realistic fused system. The 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

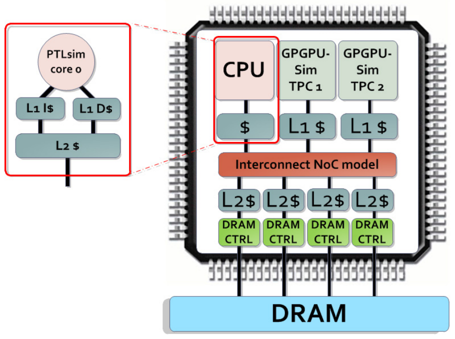

Fig.1. FusionSim's Fused model.

coherence mechanism for correct CPU and GPU communication. Using workloads from the Rodinia benchmark suite [1] 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 analytical explanation of the simulation results obtained with FusionSim. The experiments demonstrate that our simple, software-managed coherence mechanism is of low overhead for most workloads.

Fused Model: FusionSim models an x86 out-of-order CPU and a CUDA-capable GPU that operate concurrently. FusionSim correctly models ordering and overlap in time of asynchronous and synchronous operations, memory transfers, CUDA events, kernels and CPU processing. Figure 2 shows the fused system's structure. One of the Processing Clusters (PCs) of GPGPU-Sim [5] is replaced with a PTLsim [3] CPU core and its local cache hierarchy. The CPU cache hierarchy is derived from MARSSx86 [4] and interfaces with Intersim, a NoC model of GPGPU-Sim. Intersim connects the CPU cache hierarchy and the GPU TPCs 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. The LLC caches the CPU's address space and the GPU's global and local CUDA memory spaces. The CPU L1 and L2 caches and the GPU L1 caches remain private to the CPU and the GPU, respectively.

The GPU's global address space is placed in the CPU's memory address space allowing for communication between the CPU and the GPU. CPU- and GPU-private caches may both cache blocks from the global GPU space.

Conventionally, writes by the CPU or the GPU are not coherent with respect to each other. We study a simple software-based coherence mechanism that extends the CUDA API with a *cudaSelectivelyFlush* function. To make CPU changes visible to the GPU, the programmer must call *cudaSelectivelyFlush* specifying the address and the size of the memory region. This call invalidates all blocks from the region cached in the private CPU L1 and L2 caches after writing those blocks that are dirty to the last level cache and the DRAM. GPU changes are made visible by flushing the GPU's L1 caches prior to signaling kernel completion. 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.

Experimental Setup: We modeled a discrete and a fused system. Both use a single-core out-of-order CPU operating at 3.25 GHz with private 64KB L1 data and instruction caches, and a unified 256KB L2. The discrete system has a 2MB L3. All caches are write-back. The GPU is modeled after NVIDIA's Quadro FX 5800 (GT200). For the fused system the GPU's L2 is 2MB and acts as an L3 for the CPU.

We simulate ten benchmarks from the Rodinia benchmarks [1]. Execution time measurements exclude the time required for data input generation and result output. To understand the performance potential of reduced API latencies due to fusion and to appropriately model the latency of the CUDA calls we measured their latency on existing systems. We then studied a range of latencies based on these measurements. Kernel spawn latency, *KSL*, is the delay between the *cudaLaunchKernel* API function call and the actual kernel execution on the GPU. We modified the *bandwidthTest* utility from the CUDA SDK to measure the throughput of a micro-benchmark kernel and estimated the KSL on a number of actual discrete systems. KSL varied from 10 µsec to 100 µsec. We use these two values to model an optimistic and a pessimistic discrete system. Fusion will reduce KSL. Accordingly, we modeled latencies of 0.1 µsec ($\approx$300 CPU cycles), 1 µsec and 10 µsec. We measured the latency of the CUDA memory copy operations and appropriately configured the discrete model. The fused simulator appropriately models the latency of flushing data from the private caches that replaces these memory copies.

A. Analytical model of performance benefits with Fusion

We demonstrate analytically that the benefits from fusion are higher for benchmarks that a) operate on small data inputs, b) have high data throughput kernels, c) do multiple kernel launch and CUDA memory copy calls, and d) spend most of their time executing the CUDA code. Our model predicts that the speed-up from fusion reduces with the input data size. Table I lists the parameters used in our model. The kernel can be modeled as a channel of throughput $\Theta_{KER}(data_{KER})$ as the actual throughput will vary with $data_{KER}$. As $data_{KER}$ increases, Θ_{KER} saturates. For simplicity, and without the loss of generality, in the rest of the section we will simply write Θ_{KER}.

TABLE I: Analytical Model Parameters

Symbol	Description
G_{GPU}	Speed-up due to fusion of the benchmark's CUDA portion.
G_{TOT}	Total speed-up of the benchmark
n_{KER}	Number of kernel invocations
t_{GPU}	Time consumed by execution of the CUDA code
$data_{TOT}$	Total input data size of the benchmark
$data_{KER}$	Data size per kernel invocation
δ_{TOT}	Total latency of a single computation iteration
δ_{KS}	Kernel spawn latency
δ_{COPY}	Latency of the CUDA host-to/from-device memory copy
Θ_{KER}	Kernel data throughput
Θ_{COPY}	Bandwidth of the host-to/from-device memory copy

First, we develop an expression for the time t_{GPU} spent in execution of the CUDA code on a discrete system. All the applications studied do a series of the following: transfer data from CPU to GPU, compute on the GPU, transfer the results back to the CPU. For such applications t_{GPU} is described by the following:

$$t_{GPU} = n_{KER} \times \left(\delta_{TOT} + \frac{data_{KER}}{\Theta_{KER}} \right) \qquad (1)$$

where δ_{TOT} is the total latency per iteration resulting from the memory transfers and the kernel spawn. For CUDA applications that do not utilize multiple concurrent CUDA streams the total latency per single computation iteration δ_{TOT} comprises of the time spent transferring the data to or from the device and the kernel spawn latency:

$$\delta_{TOT} = 2 \times \left(\delta_{COPY} + \frac{data_{KER}}{\Theta_{COPY}} \right) + \delta_{KS} \qquad (2)$$

This expression holds for all our 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:

$$t'_{GPU} = n_{KER} \times \left(\delta'_{KS} + \frac{data_{KER}}{\Theta_{KER}} \right) \approx n_{KER} \times \frac{data_{KER}}{\Theta_{KER}} \qquad (3)$$

the speed-up of the CUDA code is given by:

$$G_{GPU} = \frac{t_{GPU}}{t'_{GPU}} \approx \frac{\delta_{TOT} \times \Theta_{KER}}{data_{KER}} + 1 \qquad (4)$$

Since $data_{KER} = data_{TOT} / n_{KER}$ and using (2) for δ_{TOT}, we obtain:

$$\boxed{G_{GPU} \approx \quad 1+\frac{\Delta_{TOTAL}}{data_{TOTAL}}\times\Theta_{KERNEL}+\frac{\Theta_{KERNEL}}{\Theta_{COPY}} \quad (5)}$$

$$G_{GPU} \approx \frac{\delta_{TOT}\times n_{KER}}{data_{TOT}}\times\Theta_{KER}+1$$

In (5), Δ_{TOTAL} is the total latency of the benchmark execution including all kernel spawn and memory transfers. The 2nd term of (5) is responsible for the speed-up due to the reduction of the total cumulative latency Δ_{TOTAL} by fusion. Intuitively, this is the number of times the GPU computation task could have been accomplished during the total latency time of Δ_{TOTAL} with infinite PCIe bandwidth. Δ_{TOTAL} is only *reduced* by fusion since KSL is not eliminated. The 3rd term of (5) is responsible for the speed-up due to the elimination of the PCIe throughput limitation. Intuitively, this is the ratio of GPU computation throughput over PCIe throughput.
Equation (5) can be rewritten as follows:

$$\boxed{G_{GPU} \approx 1+\frac{\delta_{TOT}}{data_{KER}}\times\Theta_{KERNEL}(data_{KER})+\frac{\Theta_{KERNEL}(data_{KER})}{\Theta_{COPY}} \quad (6)}$$

A kernel's throughput Θ_{KER} increases with $data_{KER}$ for small $data_{KER}$ values and saturates to a constant for large $data_{KER}$ values. The throughput saturates when the input data size is sufficient for maximum possible warp scheduler occupancy for the given kernel. Due to the $\Theta_{KERNEL}(data_{KER})$ dependency and since $data_{KER}$ is in the denominator of the 2nd term in (6) it follows that (a) the 2nd term is dominant for small $data_{KER}$ and is insignificant for large $data_{KER}$ and (b) the 3rd term is dominant for large $data_{KER}$ and is insignificant for small $data_{KER}$.

For benchmarks utilizing CUDA streams and overlapping kernel execution with data transfers the latency is bounded by (2):

$$\delta_{TOT} \le 2\times\left(\delta_{COPY}+\frac{data_{KER}}{\Theta_{COPY}}\right)+\delta_{KS} \qquad (7)$$

This results in a smaller speed-up G_{GPU} for such benchmarks. Further, by applying Amdahl's law we get an expression for the total benchmark speed-up G_{TOT}:

$$G_{TOT}=\frac{1}{\%_{CPU}+\%_{GPU}\times G_{GPU}}\le G_{GPU} \qquad (8)$$

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_{GPU}. From (5) and (8) we see that the following factors result in higher performance benefits from fusion: (i) High benchmark kernel throughput Θ_{KER}, (ii) small benchmark input data size $data_{TOT}$, (iii) many kernel invocations (large n_{KER}), and (iv) long time spent in CUDA code relative to the x86 code (large $\%_{GPU}$)

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

A. *Performance With Zero-Overhead Coherence*

We initially assume zero overhead for the software-based coherence mechanism to obtain an upper bound on performance.

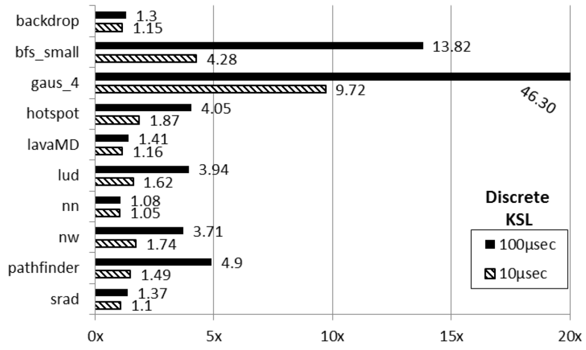

Fig.2. Fusion performance relative to discrete systems. No coherence overhead.

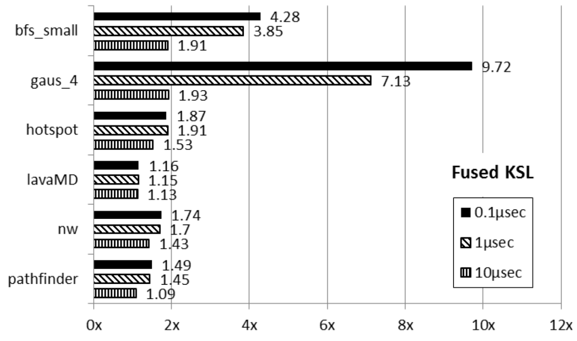

Fig.3. Speed-up of fused systems with different KSL relative to the 10 µsec latency discrete system.

Figure 2 reports the relative performance improvement of the fused system with the optimistic 0.1 µsec KSL over discrete systems with 100 µsec or 10 µsec KSL. The performance boost due to fusion varies substantially depending on the benchmark, the data input size (compare *gaus_4* and *gaus_128*), and the discrete system's KSL. The lower the discrete KSL the less the benefits from fusion. From this point our baseline discrete system will have a 10 µsec KSL.

Figure 3 reports fusion performance with KSL of 0.1, 1, and 10 µsec. We limit our attention to the benchmarks that benefited most from fusion. Fusion benefits remain mostly unaffected even when KSL is as high as 1 µsec. Even when fusion does not reduce KSL, it improves performance from 9% (*pathfinder*) to 93% (*gaus_4*). From this point on, our fused system will use an optimistic 0.1 µsec KSL.

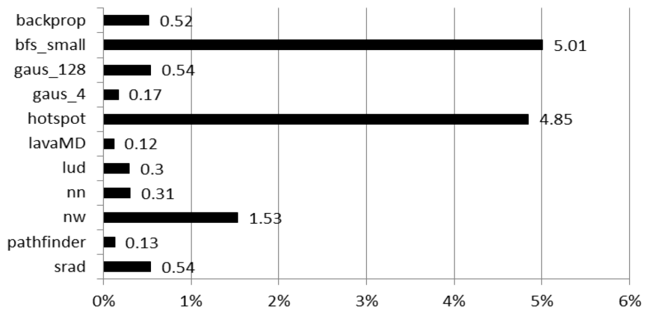

Fig.4. Performance overhead of software-based memory coherence.

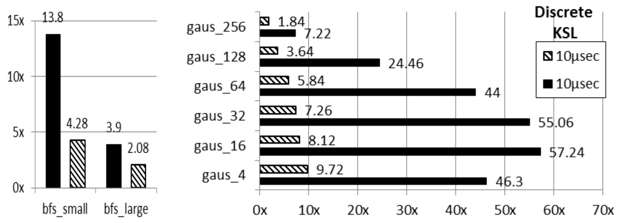

Fig.5. Data size vs. fusion performance for *bfs* and *gaus*.

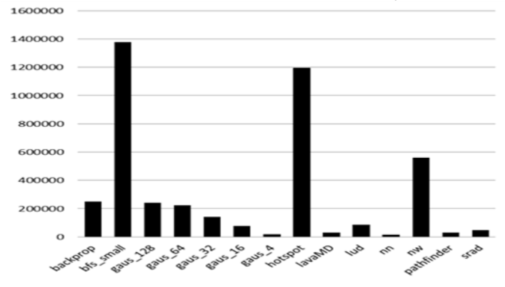

Fig.6. Kernel throughput of Rodinia benchmarks.

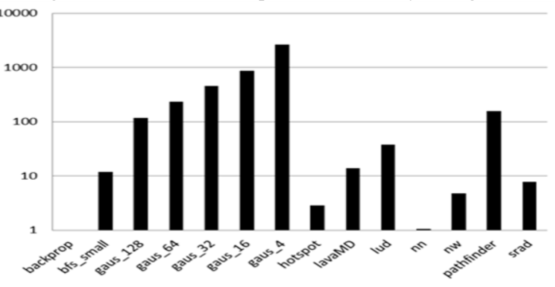

Fig.7. The $\frac{\delta_{TOT} \times n_{KER}}{data_{TOT}}$ of (5) normalized over *backprop*.

B. Performance With Software Coherence

Figure 4 shows that the memory coherence performance overhead on the fused system is less than 2% for most benchmarks and as high as 5% for *bfs_small*.

C. The Effect of the Input Data Size

We analyze how fusion benefits vary with the input data size focusing on *bfs* and *gaus* that benefited most from fusion. Figure 5 shows that fusion benefits reduce with larger input data sizes for *bfs* and *gaus*. Fusion benefits for *Gaus* drop from 9.72x with a small input to 1.84x with medium input *(gaus_256*). *Gaus_265* runs for only 10ms on the discrete system. Therefore, fusion benefits are significant with small enough input that results in about 10ms of execution time. For larger inputs, the latency of memory copy operations and the kernel spawn latency become insignificant and fusion performance approaches that of the discrete system. The fact that smaller input sets boost fusion benefits should encourage software developers to use the GPU for finer-grain tasks.

D. Results Analysis

Figure 5 showed that when used with small inputs gaus and bfs significantly benefit from fusion (9.72x and 4.28x respectively) while nn's performance improves only by 1.05x. Their respective computation patterns explain this behavior. Gauss and Bfs incur substantial latency overhead due to multiple kernel spawns inside a loop and multiple memory transfers in Bfs. Nn performs only one kernel spawn and only two memory copies accumulating only a minor latency overhead. For the same order-of-magnitude input data size the number of kernel spawns of gauss and bfs is around 100x and 16x that of nn respectively.

The analytical model (5) can be used to explain the experimental results. Figure 12 reports $\frac{\delta_{TOT} \times n_{KER}}{data_{TOT}}$ from (5) which is the effective computation latency per unit of input data (ECL). The total latency δ_{TOT} of a single computation iteration has been calculated as $(10 \times n_{KER} + n_{MEM\_CPY}) \times \alpha$, where α is some constant that gets eliminated in the normalization. The factor 10 accounts for an order-of-magnitude higher latency for kernel spawns vs. memory copy operations. To make comparisons easier, all measurements are normalized to the value of *backprop* that benefits the least from fusion.

Figure 6 shows the Bytes/sec kernel processing throughput (KPT) per benchmark. (5) predicts that fusion speed-up is proportional to KPT. *Bfs* has the highest KPT, which explains its high speed-up. *Gauss'* KPT is medium/low but gets compensated by the extremely high values of ECL in Fig. 7 thus resulting in high speed-up. *Nn* has a low KPT that additionally contributes to its low speed-up. For *gauss* and *nn* ECL is the greatest and the lowest among the benchmarks, respectively. This explains the high and the low speed-ups they experience.

References

[1] S. Che, *et al.*, Rodinia: A benchmark suite for heterogeneous computing. In IISWC, Oct. 2009.
[2] FusionSim simulation framework: www.fusionsim.ca
[3] PTLsim: cycle-accurate x86 CPU simulator. www.ptlsim.org
[4] MARSSx86: Micro-ARchitectural and System Simulator for x86 based Systems. www.marss86.org
[5] Bakhoda, *et al.*, Analyzing CUDA Workloads Using a Detailed GPU Simulator, ISPASS, April, 2009.