*i*CHAT: Inter-Cache Hardware-Assistant Data Transfer for Heterogeneous Chip Multiprocessors

Ting Cao[†] * Junli Gu Brad Beckmann

[†]Australian National University AMD Research

ting.cao@anu.edu.au {junli.gu, brad.beckmann}@amd.com

Abstract

Modern heterogeneous multiprocessors integrate CPU and GPU together to provide a boost to computational performance. With tighter integration of CPU and GPU, it is critical to share and move data more efficiently in order to leverage the computational power that a GPU can provide. Initially, DMA or PCIe devices were used to transfer data between CPU and GPU with low efficiency and little flexibility. Recently single address space and coherent cache hierarchies are being adopted in heterogeneous architectures to share data more efficiently. Thus it is becoming critical to understand the communications overheads in this new context and to improve communication efficiencies for these architectures.

This paper proposes a novel approach called *i*CHAT (*i*nter-**C**ache **H**ardware-**A**ssistant data **T**ransfer) to manage data transfer between the CPU cache and the GPU cache efficiently. The *i*CHAT technique proposed in this paper detects the communication patterns and eagerly evicts data from the owner's caches and injects that data into the requestor's caches. We implement the eager eviction technique of the *i*CHAT in a simulator based on *gem5* and an AMD in-house GPU simulator. Experimental results show that the communication related eviction traffic is reduced by an average of 40% and the total directory traffic is reduced by 8% on average.

1. Introduction

The trend in hardware is for the incorporation of domain specific accelerators in chip multiprocessors to improve performance and power efficiency. One such heterogeneous architecture integrates CPU and GPU into a single chip, examples of which are AMD's Fusion architecture (which is now named as Heterogeneous System Architecture or HSA), including APU series such as LIano and Trinity. Even though a GPU can provide an enormous speedup for data parallel applications, the communication cost between CPU and GPU can result in significant performance degradation. To effectively leverage the performance of a tightly integrated CPU and GPU, it is critically important that data is shared and moved efficiently to achieve good performance and minimal power.

The process of communication involves two major steps on hardware: firstly, data is evicted from the owner's private caches, written back to the upper level memory (extra memory copies may exist here by copying from owner's memory space to requestor's memory space); and secondly, the requester fetches the data from the upper level memory into its own cache. An example is the transfer of data from CPU to GPU where the GPU has to wait until data is transferred all the way from the CPU's cache to memory, then from memory to the GPU's cache. The communication latency is part of the execution critical path and can significantly degrade performance, especially when communication happens frequently during execution. In the past few years, communication involved latency and memory traffic have proven to be important factors affecting performance [4].

In earlier architectures DMA was used to copy data from one address space to another, this mechanism usually involved interrupting the CPU to complete the task. PCIe TPH (TLP Processing Hints) allows PCIe devices to communicate cache injection hints to the host CPU. They are specific to PCIe communication and don't necessarily generalize to shared memory based heterogeneous architectures. Some heterogeneous designs are adopting a single address space enabling the GPU to access the same virtual address space as the CPU (such as AMD Trinity) thus avoiding copying data during communication. Further heterogeneous cache coherence protocols are designed for CPU and GPU to share data through a shared last level cache, such as Intel Ivy Bridge. However, coherence latency and traffic can still degrade performance due to the complexity of coherence itself.

This paper proposes *i*CHAT technique, which stands for *i*nter-**C**ache **H**ardware-**A**ssistant data **T**ransfer, to reduce the communication latency and related cache traffic. *i*CHAT can detect and learn when communication happens and store the information about which data blocks have been transferred. Based on the knowledge learnt, the hardware can predict when the communication will happen again and inject the most frequently transferred blocks (hot blocks) to the requestor's cache ahead of time. We implement the eager eviction technique of the *i*CHAT in a simulator based on AMD's APU architecture. Experimental results show that the communication related eviction traffic is reduced by an average of 40% and the total directory traffic is reduced by 8% on average.

* This paper was done while Ting Cao was doing an internship in AMD Research.

The main contributions of this paper are: (1) This paper characterizes inter CPU-GPU communciation patterns of GPGPU application and categorizes communication data into two classes. (2) Based on the characteristics, this paper proposes *i*CHAT technique in the context of APU, to reduce the data transfer latency and total traffic for heterogeneous chip multiprocessors. In the following sections, we will first introduce the GPGPU data transfer characteristics. In Section 3, we will describe the proposed technique and also discuss some design alternatives and future improvements. At last, selected preliminary results and related work will be showed.

2. Background and Motivation

GPGPU Data Transfer Pattern We target GPGPU applications in this paper and choose Rodinia benchmark suite [3] as workloads in the evaluation. A typical GPGPU computing pattern is showed in Figure 1, which usually includes the following three phases.

- **Phase 1:** CPU prepares/initializes the data for GPU.
- **Phase 2:** GPU performs a set of computations within a loop with the CPU checking the results at the end of each iteration.
- **Phase 3:** CPU reads and post-processes the data generated by the GPU.

Data transfer occurs when switching between CPU and GPU computations. In Phase 1, CPU usually prepares the data for GPU. When switching to Phase 2, this data will be transferred from CPU to GPU to be used, which can be hundreds or even thousands of pages. We call those large amount of data as *initialization data*. Using *streamcluster* from Rodinia as an example, about 270 pages of initialization data are transferred to GPU after Phase 1. Within Phase 2, some particular data blocks are frequently transferred between CPU and GPU so that CPU can check the computing progress. The number of transfers usually depends on the number of iterations, which can vary from a few times to hundreds of times. For *streamcluster*, during Phase 2, Page ID 328 to 336 are transferred 1379 times between CPU and GPU. We call those frequently transferred pages in Phase 2 as *hot data*. When switching from Phase 2 to Phase 3, sometime a small amount of data may be transferred back to CPU again. The initialization and hot data usually take up the majority of communication data and thus are the focus of this paper.

The total amount of transferred data between CPU and GPU can be very large. Table 1 shows the amount of transferred data in number of pages between CPU and GPU for each benchmark. The same hot page transferred n times are counted as n transferred pages in the table. The amount of initialization data and hot data vary for different applications. For example, *dynproc, cell, hotpspot* and *nw* have more initialization data while *streamcluster, kmeans* and *particle* have more hot data. We can see even for Rodinia benchmarks with medium problem size, the total transferred data can be up to thousands of pages. Then for large problem size GPGPU applications or normal graphics applications, the data transfer will cause much more burden on cache controllers, directory and memory system. Even though GPU can achieve significant speedups for the kernel computing itself, with the communication overheads the final performance will be withheld from the promised potential.

```
1 CPU initializes the data
2 do {
3   GPU kernel executes and computes the data
4   CPU re-processes data
5 } while (condition)
6 CPU post-processes data
```

Figure 1. An example code of GPGPU computing pattern.

Benchmark	No. of pages	Benchmark	No. of pages
cell	513	lud	130
backprop	982	kmeans	37
hotspot	772	dynproc	515
bfs	61	mummer	2624
lava	244	nw	2021
nn	3415	streamcluster	5776

Table 1. Total data transferred between CPU and GPU in number of pages for Rodinia benchmarks medium size.

Baseline System Figure 2 shows a high level picture of the baseline heterogeneous system. CPU and GPU are integrated on the same chip. Both CPU and GPU have a private L2 cache while L3 cache is shared. L3 cache is like a on chip memory side write buffer which only holds the write back data from L2 caches. A directory based coarse grain coherence protocol similar to [10] [1] is implemented. The grain size used in this paper is page size.

Data transfers between CPU and GPU normally happen in a passive way, which means the data is only transferred on the requestor's demand. In our baseline system, when the requestor (either CPU or GPU) needs some data from the owner (GPU or CPU), it has to go through a four-hop process: first, the requestor sends a data request to the directory; second, the directory sends an invalidation request to the owner; third, the owner's cache evicts the data and writes it back to the L3 cache; and fourth, the requestor fetches the data from L3 cache into its own cache. The four-hop data transfer results in high latency and significant cache and directory traffic. When the transferred data block is large, the directory will become the bottleneck. These are the overheads of the CPU-GPU communication which degrade the performance of heterogeneous multiprocessors.

3. Inter-Cache Hardware Communicator

Based on the GPGPU computing pattern and communication characteristics, we proposes inter-cache hardware communicator *i*CHAT to move the communication data between CPU and GPU in parallel with computation to hide the commu-

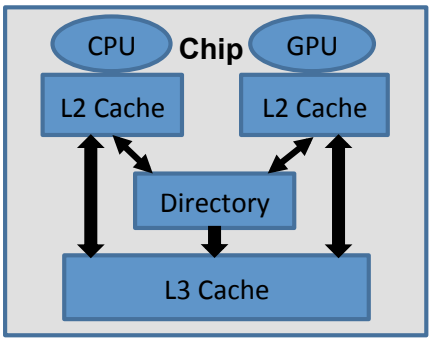


Figure 2. The baseline heterogeneous system used in the paper.

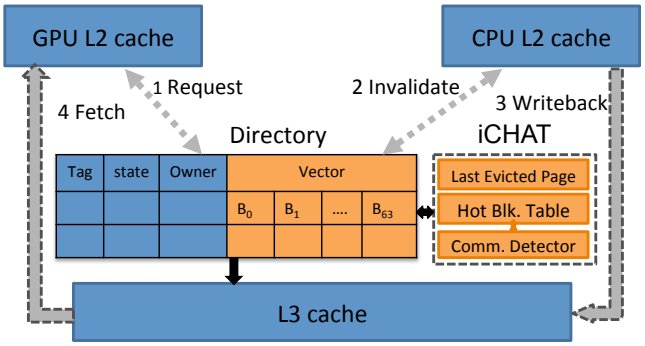


Figure 3. Integration of *i*CHAT with heterogeneous multiprocessors. Coarse grain coherence protocols are used in this paper. Each directory entry has information for each page (tag: page address, state: permission, owner: the ownership of the entry). The 1, 2, 3 and 4 are the original four hops to transfer data. With *i*CHAT, at least hop 2 and 3 will be hidden.

nication latency. The communicator detects when the GPU requests data blocks that are owned by CPU, or when CPU asks for GPU's data. Thus it needs to be integrated with the centralized memory hierarchy, which can watch both CPU and GPU data accesses as well as identifying the owner for a given address. So we connect the communicator to the directory. Figure 3 shows the integration of *i*CHAT communicator with the baseline system. It sits between L2 caches and the shared L3 cache, connecting to the directory. Thus the communicator can interact with the cache hierarchies through directory and watch the data transfers between CPU and GPU. The *i*CHAT communicator includes three components: the *communication detector* detects communication data and predicts when the communication happens; the *last evicted page* records the latest evicted page during transfers of the initialization data; the *hot block table* stores the addresses of the data blocks that are frequently transferred. The communicator transfers the communication data ahead of time through the following 3 steps. In the following sections, we will describe the mechanism of each step in details.

- **Step 1**: Capture the communication data and pattern .
- **Step 2**: Evict communication data from owner's cache to L3 cache.
- **Step 3**: Inject the communication data from L3 to requester's cache.

3.1 Communication Detection

In order to speedup the data transfer, it is critical for the communicator to be able to detect the communication data and their patterns. Communication detector captures those information by watching the traffic between CPU and GPU. The detection mechanisms for initialization data and hot data are different due to their specific characteristics. The captured information are stored in *last evicted page* buffer and *hot block table* separately.

Initializaiton Data As discussed in previous section, GPGPU computing can consume large amount of data and normally those data will be initialized by CPU in Phase 1. When switching to Phase 2, as many as hundreds or even thousands of pages of initialization data will be then transferred from CPU to GPU. Experiments show that the initialization data has specific characteristics which can be used for detection. First, initialization data is usually transferred just once during the application execution. Second, CPU usually streams through each element of the data structure to initialize it. Thus the initialization data normally consists of large amount of continuous pages with all cachelines within each page touched by CPU. We add a *validation vector* in the directory for each page as seen in Figure 3. The length of the vector is 64 bits and each bit represents one cache block of the page to indicate whether the block is in the cache. By checking the validation vector the detector could tell whether all the cachelines in the page are accessed. When GPU starts to request CPU's data, the detector will check whether the data meets the above features. If yes, the data will be identified as initialization data and the current page ID will be stored in the the *last evicted page* buffer for eager eviction.

Hot Data The basic idea to speed up hot data communication is to learn and capture the hot pages which are frequently transferred between CPU and GPU, then predict and proceed the data transfer before communication data is requested. With the proposed communicator, the hot data can be moved into the requestor (CPU or GPU) L2 cache before it asks for it. So once the requestor starts to issue the fetch instruction for the hot data, the data will be ready in the L2 cache. Thus there is no need to wait for the data to transfer all the way from the owner's side. The major benefit of this technique is that it can hide the 4 hops data transfer latency between CPU and GPU while reduce related directory and cache traffic.

When CPU and GPU are requesting each other's data, but the features of the initialization data does not apply. The data

blocks will be identified as hot data. The hot data blocks can be recorded at fine-grain or coarse-grain granularity. Page level granularity together with valid bit for each cacheline is used in this paper. The addresses of transferred data blocks are stored in the hot block table with a counter that increases each time when the block is transferred. By sorting the counters, the hot data's page ID will be identified and stored in the hot block table. Since the hot pages are normally not many, we set the hot block table to be 10 entries currently and each counter takes 3 bits.

There are a few mechanisms to detect and predict the time for communication. One straightforward mechanism is interval-based prediction. As we found out in GPGPU computing, each of the three phases showed in Figure 1 tends to have uniformed interval. In order to calculate the interval, communication detector can record each time stamp of data transfers between CPU and GPU. The recorded communication time can be local network or cache cycles. After the communication happens a few times, the detector can get an average interval length for each phase and prediction can be made when switching between phases. However, given the interval, when the communicator should start to move the data is the critical design tradeoff. An inaccurate interval based prediction might result in moving data too early (before the data is ready on the owner side) or too late (not until the requestor needs it). A more accurate detector should rely on the hardware information. One design alternative could use write-activation mechanism, which means the communicator decides that data is ready when the owner modifies the data. This mechanism will make sure that data are touched before moving back to the other side. A further last-write technique could be used in case the owner will write the data several times before data is finally ready to be transferred. Another design alternative can be requestor initialized. The communicator only starts to move the data once it sees one access request falls into the hot block table (the requestor starts to use the communication data). This technique will avoid moving the data too early that causes the cache pollution in the requestor's cache. We use the requestor initialized mechanism in the initial implementation and evaluation.

3.2 Eager Eviction

In the previous subsection, the communicator detector identifies both the initialization and hot data, and stores the information separately in *last evicted page* and *hot block table*. Eager eviction is a technique to evict the detected data from the owner's cache ahead of time. *i*CHAT conducts eager eviction by sending invalidation request (through directory) to the owner's cache which evicts the data blocks and writes back to L3 cache. Eager eviction requests will be sent when the directory is free, to avoid bandwidth conflicts with processor's data demand.

The eager eviction for initialization data is triggered when GPU starts to request the very first pages of the initialization data. Eviction of the neighbouring pages will begin when the directory is free. The *i*CHAT communicator will check the validation vector of the page with the ID number next to the last evicted page. If all the cacheline blocks of the next page are in the CPU cache, the next page will also be taken as initialization data and will be eagerly evicted. The eager eviction will stop when the features of initialization data on longer applies. Eager eviction for hot data is straightforward. When communication is predicted to start, only data blocks recorded in *hot block table* are evicted.

With eager eviction, when requestor later requests communication data, it can go to fetch data directly from L3 cache without waiting for the owner to evict the requested data. Thus eager eviction technique can transform the original four-hop data transfer into two-hop transfer. As seen in Figure 3, the hop 2 (invalidate) and 3 (writeback) will be hidden, just hop 1 (request) and 4 (fetch) are left. The traffic on directory will be reduced too.

3.3 Communication Injection

The last step for eagerly transferring the communication data is to inject them from the L3 cache to the requestor's private caches. Since the L2 cache size is normally not big compared to L3, the communication injection has to be designed to avoid cache pollution to the requestor's cache. The hot pages for GPGPU applications are usually not much (most are just one to three pages), and also will be consumed pretty soon. So the injection will not cause cache pollution by evicting other useful data. However, initialization data blocks are usually very large. In order to avoid cache pollution, a threshold based stepped injection mechanism will be used in our future work. Injection can break into three steps: first, dynamically determine an injection page number threshold based on the total number of communication data; and then inject threshold number of the pages in the requester's cache; later when the requester is starting to consume the data, inject another threshold number of the data pages to the requester's cache until all the communication data are transferred.

After the three steps of communication data eager transfer, the data will be in the requestor's private cache already when it is needed. As seen in Figure 3, the 1, 2, 3 and 4 hops latency will all be hidden and the data transfer traffic is reduced.

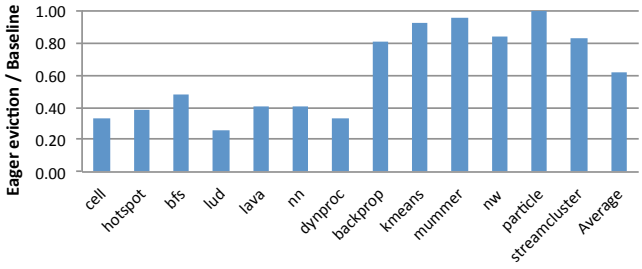
The control logic of *i*CHAT is straightforward and simple. The hardware complexity mainly lies in the storage overhead to store the communication information including page validation vector, the addresses of the last evicted page and the hot block table.

4. Experimental Results

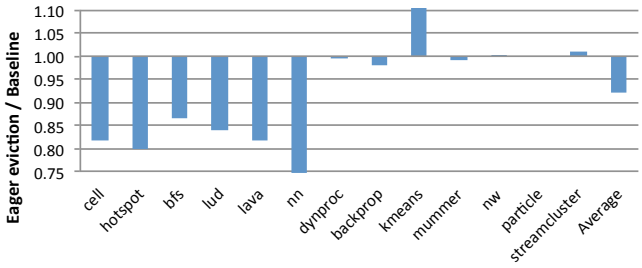
For the CPU and memory system simulation, we use the *gem5* [2] simulator system. For simulating the GPU, we use a proprietary simulator based on the AMD Graphics Core Next architecture [9]. The heterogeneous architecture

CPU clock	2 GHz
CPU core #	2
CPU L1 data cache	64 kB (2-way banked)
CPU L1 instruction cache	64 kB (2-way banked)
CPU shared L2 cache	2 MB (8-way banked)
GPU clock	1 GHz
Compute Units #	8
CU SIMD width	64 scalar units by 4 SIMDs
GPU L1 data cache	64 kB (8-way banked)
GPU L1 instruction cache	64 kB (4-way banked)
GPU shared L2 cache	1 MB (16-way banked)
L3 Memory-Side Cache	4 MB (8-way banked)

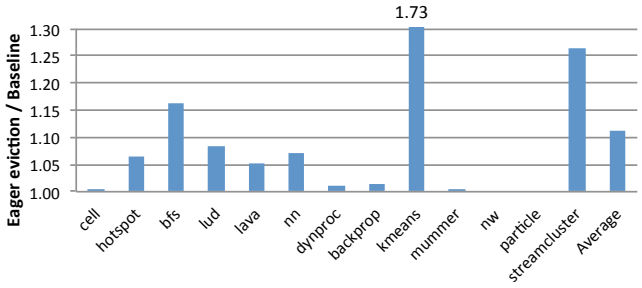
Table 2. Simulator Parameters.



(a) The eviction requests issued by the GPU.



(b) The total traffic through directory.



(c) Miss prediction rate.

Figure 4. Eager eviction technique behavior compared to baseline architecture. The eviction requests and directory traffic are reduced.

is simulated by combining the memory systems of *gem5* and the GPU simulator. The *baseline architecture* settings used in this paper are shown in Table 2.

We choose the Rodinia benchmark suite with a medium problem size as the heterogenous workloads in the evaluation. In this paper, we evaluate the result of applying the proposed technique for initialization data. The evaluation for hot data transfer will be showed in a future publication.

As the eager eviction technique can evict the data from CPU L2 cache to L3 cache in advance, the eviction requests will be reduced. Successful eager eviction of a page enables the GPU to fetch that page directly from L3. So the 4-hop data transfer process will be reduced to 2 hops as explained in Section 3.2. Figure 4(a) shows the total eviction requests issued by GPU compared to the baseline. Eager eviction reduces the eviction traffic significantly, up to 74% for *lud* and 40% on average. The results clearly fall into two groups. For the first seven benchmarks on the left side, the eager eviction can reduce traffic by 60% on average. While for the rest six benchmarks on the right side, the traffic is reduced less than 20%. The first reason for that is some of the six benchmarks are scientific benchmarks without many pages transferred. For example, *kmeans* just has five pages of initialization data and *particle* just has one. Thus the potential space of optimization for those benchmarks is not much. Another reason is that our technique eagerly evicts pages in an incremental sequence of page ID starting with an ID of the last evicted page. However, if GPU has a scattered data access footprint, it may request a page before the communicator has a chance to evict it. In this situation, GPU still needs to go through the original 4-hop data transfer process.

As the directory does not need to send invalidation requests to the owner, the total traffic seen by the directory is also reduced. As showed in Figure 4(b) depending on how much eviction traffic counts in the total traffic, the total traffic decreases from a few percent to 25% for *nn*, and 8% on average. It is also separated into two groups for the similar reasons as Figure 4(a). The total traffic for *kmeans* increases. This is because of the miss predicted eviction, which results in more data blocks being evicted than necessary. Figure 4(c) shows the miss eviction rate for each benchmark. The miss rate for *kmeans* is specially high resulting in an increase in eviction traffic as well as extra requests to bring the miss evicted data back from L3 cache into CPU L2 cache. However, since the initialization data is not much, just five pages for *kmeans*, the absolute value of traffic increase won't be that much. For the other benchmarks with relatively high miss prediction rate, such as *bfs*, the basic reason is that the eager eviction evicts some non-initialization data which also meets the features of initilization data and has continuous page ID following the last evicted page. One way to fix this problem is to set a threshold for the number of pages can be evicted each time, such as 5, to slow down and double check whether GPU consumes the data.

5. Related Work

Eager Eviction There are several eager write-back schemes proposed before to reduce shared data coherence overhead or increase row hits for CPU. However, none of them are designed for inter CPU-GPU data transfer characteristics.

Lebeck et al. [7] propose the first speculative invalidation technique, called Dynamic Self-Invalidation, for cache-

coherent distributed shared memory system. The idea of this scheme is based on the observation that data blocks that have recently had conflicting accesses—and hence would have needed invalidation—are candidates for self-invalidations. To predict when to self-invalidate a block, they use synchronization boundaries to trigger block self-invalidation. The technique can predict when a processor completes accessing a shared block and speculatively invalidate the block in advance so that subsequent accesses by other processors can be fastened. For similar purpose, Lai et al. [6] propose Last Touch Predictors. It is based on the observation that memory sharing and invalidation are triggered by program instructions. The technique maintains an instruction trace from a coherence miss until last touch to a block before invalidation. As the program behavior is repetitive, it is possible to use this trace to predict block invalidation. Stuecheli et al. [14] propose Virtual Write Queue scheme for eager write-back to increase row-level access locality. While write operations in the DRAM write queue are being scheduled, other dirty cache lines that mapped to the same row as the scheduled ones are searched in the last-level cache and immediately transferred to DRAM.

Compared to the existing solutions, our proposed *i*CHAT communicator detects the hot regions transferred between CPU and GPU, and eagerly evicts them from the owner's cache. The region-level invalidation can better match the large data needs for GPU applications and reduce the cache and directory traffic.

Cache Injection Cache injection was proposed to speedup producer-consumer style communication on distributed parallel machine [11]. Once producer produced new data, it would be sent and injected to the requestor's cache. One example for CPU cache injection technique is [5]. Cache injection technique has not been applied to GPU related platform. The hardware or software prefetchers [13] [12] [8] serve similar purpose as injection, reducing data accessing latency. However, the prefetching techniques may not cover communication data since communication data has different locality. Our injection technique can be used to supplement prefetching techniques to speedup communication data access.

6. Conclusions

The frequent transfers of large data blocks between CPU and GPU result in high latency and heavy cache and memory traffic, which withhold the heterogeneous multiprocessors from potential speedups. This paper proposes a technique called *i*CHAT (inter-Cache Hardware-Assistant data Transfer) to watch and detect the communication pattern, eagerly evict the communication data from the current owner's caches and inject it into the requestor's caches ahead of time. Using *i*CHAT can reduce cache traffic and the long latency involved in CPU and GPU communication, and thus enhance the performance boost by leveraging GPU as hardware accelerator.

References

[1] M. Alisafaee. Spatiotemporal coherence tracking. In *45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 341–350, 2012.

[2] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC 2009)*, pages 44–54. IEEE, 2009. ISBN 978-1-4244-5156-2.

[4] C. Gregg and K. M. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 134–144, 2011.

[5] R. Huggahalli, R. R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network i/o. In *32st International Symposium on Computer Architecture*, pages 50–59, 2005.

[6] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 139–148, 2000.

[7] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, 1995.

[8] J. Lee, N. B. Lakshminarayana, H. Kim, and R. W. Vuduc. Many-thread aware prefetching mechanisms for gpgpu applications. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 213–224, 2010.

[9] M. Mantor and M. Houston. Amd Graphic Core Next. In *AMD Fusion dedeveloper summit*, 2011.

[10] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *Proceedings of 32st International Symposium on Computer Architecture*, pages 234–245, 2005.

[11] T. C. Mowry. *Tolerating latency through software-controlled data prefetching*. PhD thesis, Stanford University, May 1994.

[12] NVIDIA. NVIDIA next generation CUDA compute architecture: Fermi, 2009.

[13] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. mei W. Hwu. Program optimization space pruning for a multithreaded gpu. In *Sixth International Symposium on Code Generation and Optimization*, pages 195–204, 2008.

[14] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The virtual write queue: coordinating dram and last-level cache policies. In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 72–82, 2010.