Design Space Exploration of Memory Model for Heterogeneous Computing

Jieun Lim
School of Electrical and
Computer Engineering
Seoul National University
Email: Jelim@dsp.snu.ack.kr

Hyesoon Kim
School of Computer Sciencne
Georgia Institute of Technology
Email: hyesoon@cc.gatech.edu

***Abstract*—Heterogeneous computing that combines a traditional CPU architecture with an accelerator has become a popular architecture. Memory modeling design decisions affect not only architecture designs but also programming models. Hence, comparing them is very challenging and not all design spaces have been explored. Although a unified memory address space that is fully coherent and strongly consistent across the entire memory system would be the ideal case, because of scalability and complexity, less ideal designs have been proposed.**

In this paper, we explore various design options quantitatively and qualitatively. Our results suggest that maintaining a separate memory model for each architecture and having a partially shared memory space provide the most design options in both programming models and architecture designs.

I. Introduction

Heterogeneous computing has become one of the major architecture design trends. In particular, combining CPUs and GPUs is a popular option as demonstrated by AMD's Fusion [3], Intel's Sandy Bridge [17], IBM Cell processors [16], and NVIDIA's Denver [28]. Although these heterogeneous computing models have different ISAs and different execution cores, they share some levels of memory systems to reduce the communication cost.

In a heterogeneous system, there are several ways to design memory systems, from completely disjoint memory systems (e.g., connections through PCI-E) to fully shared unified memory systems. Although a unified memory address, fully coherent and strongly consistent memory system can provide the most convenient programming environment, because of hardware cost and complexity, various options have been proposed. Furthermore, separate memory systems for each architecture could also reduce design and verification time.

A memory model is strongly coupled with architecture design and programming models, which makes it more difficult to compare models among different design options. Furthermore, these memory model design options must be studied together in the early architecture and ISA design stage.

Hence, the goal of this paper is to understand the trade-offs in memory system design decisions. We aim to evaluate the overhead of programming model decisions and the hardware design options separately. communication mechanisms as the major memory system design parameters. We also introduce several hybrid locality management schemes that allow the system to maintain its own locality management scheme in the shared address space. Among these parameters, we quantitatively evaluate memory address space options and hardware communication methods.

Our results suggest that maintaining a separate memory model for each architecture and having a partially shared memory space provide various hardware and programming model design options. For example, the architecture can have an explicit or implicit locality management scheme to accelerate its specialized applications. A partially shared address space provides opportunities to optimize private address spaces independently.

In summary our work makes the following contributions:

1) We explore the design options of memory systems in heterogeneous computing by considering both programming models and hardware designs.
2) We propose several hybrid locality management schemes in programming models for heterogeneous computing and also the corresponding hardware mechanisms.
3) Finally, our results show that a partially shared address space scheme provides the most versatile design options in locality management and communication methods. This allows for various architecture and programming model design options.

II. Memory Design Options for Heterogeneous Computing

Heterogeneous computing is typically a combination of different accelerators and general purpose processors. Here, we choose CPUs to represent general purpose processors and GPUs for accelerators. However, all the discussions and studies can be applied to other accelerators such as DSPs or other ISAs. We use the term *processing unit (PU)* to refer to either CPUs or GPUs.

Since our focus is not discussing memory design options for the private memory space in each PU, we assume that each PU has its own memory model.

A. Memory Address Space Design Options

The design options of a memory address space are unified, partially shared, disjoint, and asymmetric distributed shared (ADSM), as illustrated in Figure 1. The disjoint memory space has been the most widely used in heterogeneous computing until now because each PU simply has its own private memory space without the need to maintain coherence and consistency between memory space. Please note that here we are only discussing memory *address* space.

1) Unified Memory Address Space: A unified memory address space means that there is no separation between CPU address space and GPU address space. Any tasks can be run on any PU without explicit data transfer commands. Programmers can easily execute libraries and existing program paradigms that are developed for homogeneous computing systems.

Unified Memory Space $\neq$ Coherence Support: As can be seen in an example from CUDA 4.0 [29], a unified memory space can have non-coherent memory space between CPUs and GPUs.

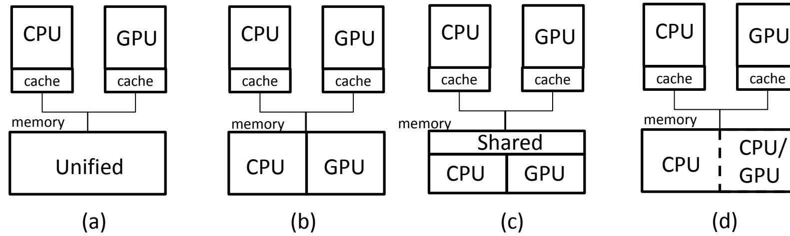

Fig. 1. Memory address space design options: (a) unified memory address space, (b) disjoint memory address space, (c) partially shared memory address space, (d) asymmetric distributed shared memory (ADSM) address space.

Implementation: A unified memory address space can be implemented as a virtually unified memory address space (like CUDA 4.0) [29], [6] or as a physically unified memory. When it shares only virtual addresses, one memory address space maps to different physical addresses on each PU, so it can work in both discrete and integrated memory systems. This provides different page size options to each PU (e.g., GPUs can have large page size to accommodate high stream locality.) and also a different page table format to optimize for its own PU. However, having different page table formats complicates TLB designs and memory management units [31], [34]. If only a virtual address space is shared between PUs, software support such as using libraries or runtime is required to hide latency.

2) Disjoint Memory Space: In a disjoint memory address space, there should be explicit communication between two address spaces in order to access data allocated in the other space. This minimizes the cost of designing/building heterogeneous computing systems and also provides scalability but it increases the burden on programmers.

Implementation: Again, address space and physical memories are orthogonal. Physical memories can be implemented using any network fabric such as PCI-E, DMA, and interconnection network. Disjoint memory systems can have either the same or different page table formats between address spaces. Explicit communication is always required.

Disjoint Memory Space $\neq$ Non-shared Cache: Even though memory spaces are not shared, they can still share the cache. Intel's Sandy Bridge is an example [17]. CPUs and GPUs have discrete memory spaces, but they share the last-level cache for better resource management.

3) Partially Shared Memory Space: Finally, in a partially shared space, a part of the memory space is shared to get benefits from both the convenience of using shared memory and to reduce the hardware design cost. This scheme requires additional overhead if managing between shared space and non-shared space.

This partially shared memory space introduces several different design options such as the concept of ownership and different locality management.

Ownership Control: In this scheme, even though a subset of address space is shared, each PU has ownership. This prevents the address space from being updated by both PUs concurrently. Hence, the shared memory address space does not need to maintain coherence. This ownership is introduced in Intel's heterogeneous architecture programming model [31]. We refer to this mechanism as the LRB programming model. Ownership is for performance optimizations and is not essential for partially shared memory space design.

The downside of having this ownership is that either programmers or compilers have to insert `acquire` and `release` ownership commands. It also needs to specify whether or not data objects are allocated in the shared memory space. UPC[2] and LRB used the `shared` type qualifier to handle this issue. Unfortunately, it is the programmer's responsibility to tag all data shared between the CPUs and GPUs with the `shared` keyword. Globalization and privatization can also be performed during program execution to indicate ownership changes.

Figure 2 shows reduction examples from two different programming models: unified and partially shared address spaces.[1] These examples illustrate the differences of using different `malloc` and ownership changes for objects in the shared space. The initial data is loaded from the CPU and sent to the GPU. The final sum is also calculated inside the CPU.

Implementation: Although a partially shared address space can be implemented with one unified physical memory system, to get the benefits of discrete memory space (non-shared space), architectures often employ different physical memory systems. However, implementing a partially shared memory address space requires maintaining page table mapping in both CPUs and GPUs, which could generate high overhead. Note that the unified memory address space also has the same problem if both CPUs and GPUs have a virtually unified memory address space with discrete memories. To reduce this overhead, using a PCI aperture[2] was proposed [34], [31], [6]. Allocating a portion of the PCI aperture space to the user space of an application provides a common buffer between CPUs and GPUs. The PCI aperture already supports asynchronous copy between CPU memories and GPU memories. This method enables very low-cost communication between CPUs and GPUs. However, this method is intended to support only small portions of memory space between CPUs and GPUs although in principle the address space can grow dynamically.

4) Asymmetric Distribute Shared Memory Space: Another form of a partially shared memory space is an asymmetric shared memory space. While one PU can access the entire memory address space, the other PU can only access its private memory address space. Gelado et al. proposed an asymmetric distributed shared memory (ADSM) system [10]. In their proposal, CPUs can access the entire memory space but GPUs can only access their private address space. The major benefit of this mechanism is that only the CPU needs to maintain coherence and consistency while GPUs still have a simple memory system. Similar to the unified memory space, coherent data can be maintained by hardware coherence or

[1]This example is based on the programming model from the LRB architecture [31].

[2]PCI aperture is a virtual memory space where a graphics driver will initiate access to the graphics memory. It originates from the AGP aperture, which allows graphics drivers to access part of the system memory directly to copy texture and polygon meshes, and other data can be loaded directly from the CPU's physical memory to utilize direct memory access. [18].

```
int addTwoVectors(int *a, int *b, int *c)
{
  for (i = 1 to 64) {
    c[i] = a[i]+b[i];
   }
}

int kernel(...)
{
   int *a = malloc(...);  int *b = malloc(...);
   int *c = malloc(...);  int *d = malloc(...);
   int *e = malloc(...);  int *f = malloc(...);

    for (i=1; i< 64; i++) // initialize
    {
    // initialize a, b, d, e
    }
    addTwoVectors (a, b, c);   // c = a+b
    addTwoVectors (d, e, f);   // f = d+e
    addTwoVectors (c, f, f);   // f = c+f
    ....
}
             (a)

attribute(GPU)
int addGPUTwoVectors(shared int *a,
    shared int *b, shread int *c)
{
  acquireOwnership(a,b,c);
  for (i=1 to 64) {
   c[i] = a[i] + b[i];
  }
  releaseOwnership(a,b,c);
}

int kernel(...)
{
  // allocate in shared region
   int *a = sharedmalloc(...);
   int *b = sharedmalloc(...);
   int *c = sharedmalloc(...);

   int *d = malloc(...);
   int *e = malloc(...);
   int *f = malloc(...);

 for (i=1; i< 64; i++) // initialize
   {
   // initialize a, b, d, e
   }
  releaseOwnership(a, b, c);
  addGPUTwoVectors(a,b,c); // c = a+b in GPU
  addTwoVectors(d,e,f);   // f = d+e in CPU
  acquireOnwership(c);
  addTwoVectors(c,f,f);   // f = c+f in CPU
  ...
}
        (b)
```

Fig. 2. Reduction code examples in (a) unified memory space and (b) partially shared memory space using ownership optimization.

purely by software coherence support. The greatest benefit of this option is that it still provides a shared address space with discrete memories. The code examples in Figure 3 illustrate the differences between disjoint memory address space and ADSM. The difference over the unified memory address space is that ADSM uses a special memory allocation function, `adsmAlloc`, to allocate data into the shared memory space. Unlike the disjoint memory address space, there is no need to transfer data back to the host memory space.

Implementation: The basic mechanism is that the shared memory address space should be allocated in both memory systems over the same range of virtual memory addresses. In ADSM, two identical memory address ranges have been allocated to each PU. Only one PU is responsible for maintaining coherent data states using either cache coherence or a runtime system. ADSM was proposed only for software solutions using APIs. Four fundamental APIs, shared-data allocation, shared-data release, kernel invocation, and return

```
int kernel(...)
{
    // initialize a, b, c, d, e, f  in CPU
   int *a = malloc(...);  int *b = malloc(...);
   int *c = malloc(...);  int *d = malloc(...);
   int *e = malloc(...);  int *f = malloc(...);

   // duplicated pointer for a, b, c
   int *gpu_a, *gpu_b, *gpu_c;

   // allocate mem space in the GPU
   int gpu_a = GPUmemallocate (gpu_a, gpu_b, gpu_c);

   // send data from CPU to GPU
    Memcpy(gpu_a, a, MemcpyHosttoDevice);
    Memcpy(gpu_b, b, MemcpyHosttoDevice);

    addGPUTwoVectors (a, b, c);   // c = a+b;
    addTwoVectors (d, e, f);   // f = d+e

   // send data from GPU to CPU
    Memcpy(gpu_a, a, MemcpyDevicetoHost);


    addTwoVectors (c, f, f);   // f = c+f

   // free space in both CPU and GPU
    ....
}
          (a)

int kernel(...)
{

   // initialize a, b, c, d, e, f
   int *a = malloc(...);  int *b = malloc(...);
   int *c = malloc(...);  int *d = malloc(...);
   int *e = malloc(...);  int *f = malloc(...);

   // no duplicated GPU pointers
    a = adsmAlloc (64B);   // allocate mem space in the GPU
    b = adsmAlloc (64B);   // allocate mem space in the GPU
    c = adsmAlloc (64B);   // allocate mem space in the GPU

    copyfromCPUtoGPU(a,b,c);  // send data from CPU to GPU
    addGPUTwoVectors (a, b, c);   // c = a+b;

    addTwoVectors (d, e, f);   // f = d+e

    addTwoVectors (c, f, f);   // f = c+f
    accfree(a); accfree(b); accfree(c);
    ....
}
          (b)
```

Fig. 3. Reduction code examples in (a) disjoint memory space and (b) ADSM.

synchronization are essential. Typically ADSM is ideal for the relaxed consistency model so that the deadline of the data synchronization point is at the end of the kernel execution.

ADSM is very similar to the partially shared memory address space. It also has the concept of ownership. The main benefit over the partially shared memory address space is that ADSM can simplify a memory system of one PU (mainly the accelerator). However, consequently, only one PU can use the entire memory space. This design space is the ideal for asymmetric computing power(e.g., powerful CPU + low-cost accelerators).

B. Locality Management

Locality management in caches is another important memory system design parameter. Although it can be applied to all levels of the storage, we focus our discussion on the shared memory space specifically in the second-level caches.[3] For private memory space,

[3]We assume that at least 2-level cache hierarchy for this study.

each PU can have its own memory model. Hence, we exclude the disjoint memory address option in this section because naturally it has only private caches.

```
push (a, CPU.P);   // private-CPU
push (b, CPU.P);   // private-CPU
push (d, GPU.P);   // private-GPU
push (e, GPU.P);   // private-GPU
addGPUTwoVectors(a,b,c); // c = a+b in GPU
addTwoVectors(d,e,f);    // f = d+e in CPU
push (c, S);       // second-level
push (f, S);       // second-level
addTwoVectors(c,f,f);   // f = c+f in CPU
        (a)

push (CPU.a, CPU.P); // private-CPU
push (CPU.b, CPU.P); // private-CPU
push (GPU.d, GPU.P); // private-GPU
push (GPU.e, GPU.P); // private-GPU
addGPUTwoVectors(CPU.a,CPU.b,Shared.c);
addTwoVectors(GPU.d,GPU.e,Shared.f);
push (Shared.c, S);       // second-level
push (Shared.f, S);       // second-level
addTwoVectors(Shared.c, Shared.f, Shared.f);
        (b)


// transfer d, e into the GPU space
addGPUTwoVectors(CPU.a,CPU.b,Shared.c);
addTwoVectors(GPU.d,GPU.e,Shared.f);
push (Shared.c, S);       // second-level
push (Shared.f, S);       // second-level
addTwoVectors(Shared.c, Shared.f, Shared.f);
        (c)
```

Fig. 4. Reduction code examples of managing localities in (a) unified memory space, explicit-private-explicit-shared, (b) partially shared memory space, explicit-private-explicit-shared, (c) partially shared memory space, implicit-private-explicit-shared. This example assumes that the second-level caches are either physically shared between PUs or maintained by cache coherence.

We classify locality management in the shared memory space as the following three options: all implicit, all explicit, and implicit+explicit hybrid. Implicit management can be done by hardware caching, a compiler or a runtime system. Explicit management requires programmers (although this can be done by compilers as well) to explicitly control locality. Interestingly, several combinations are possible depending on the locality management scheme of the private cache: implicit-private-implicit-Second, explicit-private-explicit-Second, implicit-private-explicit-Second, etc. Although one PU can also have two locality management schemes such as in CUDA 4.0 (it has hardware caches and software managed caches), we consider one locality management scheme for each PU.

When both private space and shared space have the same locality management scheme, implementations are fairly straightforward. However, when private and shared spaces have different management schemes, additional hardware or software implementation support is required. So, we discuss only these cases. To simplify the discussion, we assume that explicit management is done by programmers and implicit management is done by hardware. Figure 4 shows the previously introduced reduction code with locality control statements. `push` explicitly places data into the desired cache hierarchy. Figure 4(c) does not have explicitly controlled locality for the primary caches.

1) Implicit-Private-Explicit-Shared: In this design option, the hardware manages the locality implicitly for private caches, but for the shared memory space, there is an explicit management from programmers. For the partially shared memory space, each PU has to explicitly send/copy data to/from the shared memory space. Hence, it is relatively easy to control the locality together. However, for the unified memory space, potentially all the memory space can belong to the shared memory space. Hence, this option is not desirable since it needs explicit management for shared data structures. The benefit of this option over the implicit-private-implicit-shared option is that it provides a locality management option without significantly increasing the programmer's burden. It is already a programmer's responsibility to decide when and which data have to be located in the partially shared memory space.

2) Explicit-Private-Implicit-Shared: This option is the opposite case of the previous design option. Although the locality of private caches is explicitly managed, the locality of the shared memory space is only implicitly managed. The benefit over the explicit-private-explicit-shared option is that the size of the shared cache is not coupled with heterogeneous computing configurations. The cache size for the shared memory address region can be easily extended by simply connecting more caches. When CPUs and GPUs are in the same chip, the shared cache size is fixed for each system from the fabrication time. Unlike implicit-private-explicit-shared, the unified shared address space can easily have this option.

3) Implicit-Private-Explicit-Private-Explicit-Shared: In this option, all private caches have different locality management schemes, but the shared memory space has the same explicit locality management scheme. This is very similar to the implicit-private-explicit-shared scheme except that CPUs and GPUs can have different management schemes for their own private memory space.

4) Implicit-Private-Explicit-Private-Implicit-Shared: The private caches have different locality management schemes just like those in Section II-B3, but the shared memory space is implicitly managed. This is also very similar to the explicit-private-implicit-shared option. For the PU with an explicit-private cache, cache hits for the shared memory space cannot be guaranteed. So, an application should immediately copy data to the private space if it wants to maintain cache hit latency.

5) Hybrid Locality in the Second-Level Cache: In this design option, all PUs have different locality management schemes and the shared memory address space itself also supports both implicit and explicit locality management. Among the possible combinations, we discuss only the case where the implicit-private PU has an implicit managed shared address space and the explicit-private scheme explicitly manages the shared space.

Hardware Implementation: If both CPUs and GPUs share a cache, the cache replacement policy has to be changed. An implicitly-managed cache block cannot evict an explicitly-managed cache block. To support this feature, (1) the tag storage has one bit to indicate the locality information to be compared in the replacement logic; and (2) the explicitly managed cache size must be smaller than the total size of the physically shared cache. This option eliminates the hardware limitations of shared cache sizes, since extra shared cache sizes can be simply managed using an implicit locality option and only critical data needs to be managed explicitly.

6) Summary: The partially shared address space provides the most options to control the locality of caches. Although several programming languages provide various options of locality management schemes, such as Sequoia, which strictly enforces locality [7], no programming models specifically design to provide locality control in heterogeneous programming platforms.

III. Summary Of Existing Heterogeneous Computing Memory Systems

Table I shows a summary of previously proposed heterogeneous computing systems and their memory systems. Just to compare, we also include one homogeneous architecture, Rigel. Typically one architecture has one programming model. However, CUDA 4.0 and CPU+CUDA* have the same hardware architecture. We also include CUDA 4.0 and OpenCL, which are programming languages, so some hardware configurations can be varied.

The summary shows that none of the heterogeneous computing systems has employed a unified, fully-coherent, strong-consistent memory system yet. Most proposed/existing systems have disjoint memory systems so locality management is only for private spaces. Currently, only CUDA 4.0 provides the unified memory address space, but it does not provide any locality management for the shared space.

IV. Evaluation Methodology

A. Simulation Infrastructure

We use the MacSim simulator [1], a cycle-level and trace-driven simulator for our simulations. We generate traces for CPUs and GPUs separately. Library or operating system effects are modeled with special instructions. Table II shows the system configuration. Our baseline processor is similar to Intel's Sandy Bridge [17], but we model a GPU core similar to NVIDIA's Fermi [27]. [4] Note that we model the cache latency based on CACTI 6.5 [15].

B. Benchmarks

We use six kernels to evaluate key components. The characteristics of the evaluated kernels are explained in Table III.

Ideally, we like to divide the work between CPUs and GPUs intelligently so that the total execution time can be minimized. Since determining how to partition the work is beyond the scope of our work ([25], [11] present sophisticated algorithms to find the optimal partitioning point), we simply divide the computational work evenly. The input data is allocated in CPUs initially and after the GPUs finish the work, data have to be transferred to the CPU.

C. Different Programming Model Effects

To model different programming model effects, we use a series of special instructions. By varying the latency of these operations, we also explore the overhead of communication methods. Table IV shows the name of the parameters and the default execution latency used in Section V-A. `api-pci` models APIs using PCI-E for communication. `api-acq` and `api-tr` model data transfer functions in the partially shared address space. `lib-pf` models the cost of handling page faults in discrete memory addresses.

V. Results

A. Case Studies

We select five distinct heterogeneous computing systems to evaluate the performance effects of the memory systems: CPU+GPU(CUDA) [29] , LRB[31], GMAC[10], and Fusion[3] and IDEAL-HETERO for a unified, fully coherent system. CPU+GPU has the disjoint memory space connected with PCI-E. LRB uses the partially shared address space with the PCI aperture. GMAC has the ADSM space also connected with PCI-E. Fusion has the disjoint memory space with a memory controller connection. To isolate any architecture effects other than the memory systems, all systems have the same CPUs and GPUs.

We divide the execution time into three categories: sequential, parallel, and communication. As shown in Figure 5, the majority of execution time is spent on parallel computation. Reduction (1.3%), merge sort (12%), and k-mean (7.6%) have relatively high communication overhead.

For CPU+GPU, once the GPU finishes computation, the final data has to be transferred to the CPU memory address space, so there are

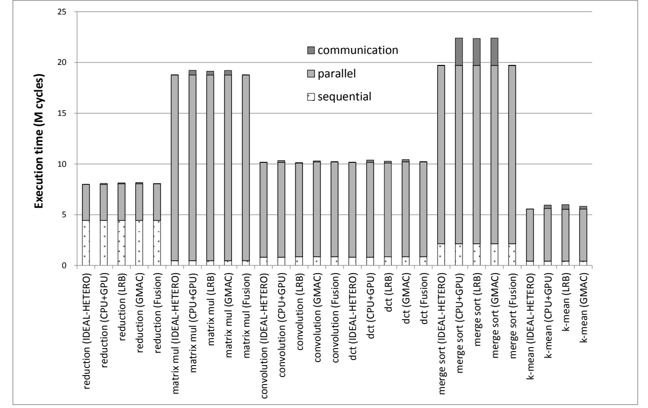

Fig. 5. Evaluation of five heterogeneous architecture configurations.

additional data transfer costs. In contrast, for LRB, if data is already located in the shared address space, transferring is not required. It still has the overhead of communication when data is initially transferred from CPUs. It also generates page faults if data in the shared space is first-time accessed.

For GMAC, asynchronous copies are performed during computation, so the communication cost can be easily hidden. For fusion, the communication is through memory controllers, so it generates memory accesses for all data transfer between CPUs and GPUs. However, the memory access cost is also very small compared to that of PCI-e. Hence, CPU+GPU, LRB and GMAC have a longer execution time than those of IDEAL-HETERO and Fusion. Figure 6 shows only the communication cost.

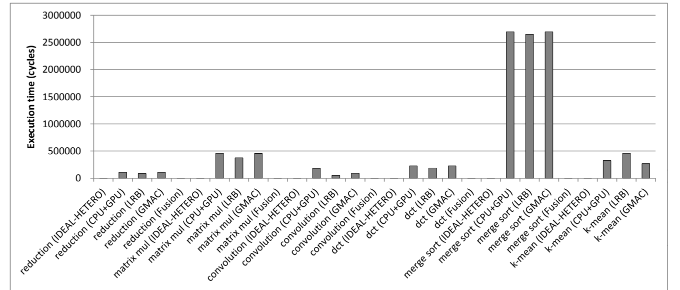

Fig. 6. Communication overhead for the evaluated heterogeneous computing.

B. Memory Space Effects

To isolate memory space effects, we assume that all the systems share the cache in this section. The four memory spaces in Section II-A, unified (UNI), disjoint (DIS), partially shared (PAS), and ADSM, are evaluated. Figure 7 shows the results. As we easily expect, there is almost no performance difference between options since additional instructions to transfer data between memory spaces are very small compared to the amount of computation. The conclusion of this experiment is that the memory address space design itself does not affect performance. It is more about programmability. The performance delta of the previous heterogeneous computing systems (Section V-A) is mainly caused by the hardware communication mechanisms.

C. Programmability vs. Memory Address Spaces

Different programming options affect how easy/difficult it is to write programs. Similar to studies in [32], [8], [5], we also use the

[4] Since we are only interested in memory systems, the number of cores in the CPU and GPU is simplified to one.

scheme	address space	Connection	coherence	how to use shared data	consistency	synchronization	Locality
CPU+CUDA* [29]	disjoint	PCI-E	-	NA	weak consistency	-	impl-pri-expl-pri
EXOCHI [34]	unified	Memory controller	can be coherent	CHI runtime API	weak consistency	unknown	impl-pri
CPU+LRB [31]	partially shared	PCI-E	coherent only in LRB/CPU	type qualifier, ownership	weak consistency	APIs	impl-pri
COMIC [21]	unified	interconnection	directory	COMIC API functions	centralized release consistency	barrier function	expl-pri-impl-pri-impl-shared
Rigel [19]	unified	interconnection	HW/SW	global memory operation	weak consistency	implicit barrier/Rigel LPI	expl
GMAC [10]	ADSM	PCI-E	GMAC protocol	global memory operation	weak consistency	sync API	expl-private-impl-shared
Sandy Bridge [17]	disjoint	Memory controller	-	-	weak consistency	-	impl-priv-exp-priv
Fusion [3]	disjoint	Memory controller	-	-	-	-	-
IBM Cell [16]	disjoint	interconnection	-	-	weak consistency	-	expl-pri-impl-priv-impl-shared
Xbox 360 [4]	disjoint	cache/FSB	-	Lock-set cache, copy	-	-	impl-priv-exp-shared
CUBA [9]	disjoint	BUS	-	direct access to local storage	weak consistency	-	exp-priv
CUDA 4.0	unified	-	-	explicit copy	weak consistency	-	exp-priv
OpenCL	unified	-	-	explicit copy	weak consistency	-	exp-priv

TABLE I. HETEROGENEOUS ARCHITECTURE SUMMARY. (CUDA* MEANS CUDA BEFORE CUDA 4.0), COMIC ASSUMES THE IBM CELL ARCHITECTURE. THE IBM CELL INDICATES THE CELL SDK.

		CPU	**GPU**
Execution core	# cores	1	1
	Execution engine	3.5GHz, out-of-order	1.5GHz, in-order, 8-wide SIMD
	Branch predictor	gshare	N/A (stall on branch)
Memory	L1	8-way 32KB L1 Dcache (2-cycle) 8-way 32KB L1 Icache (2-cycle)	8-way 32KB L1 Dcache (2-cycle) 4-way 4KB L1 Icache (1-cycle) 16KB s/w managed cache
	L2	8-way 256KB L2 Cache (8-cycle)	N/A
	L3	32-way 8MB L3 Cache (4 tiles, 20-cycle)	
Interconnection		Ring-bus network	
DRAM		Model DDR3-1333. 4 controllers, 41.6GB/s Bandwidth, FR-FCFS	

TABLE II. BASELINE SYSTEM CONFIGURATION.

Name	compute pattern	# of instructions			# of communications	initial transfer data size (B)
		CPU	GPU	serial		
reduction	parallel → merge → sequential	70006	70001	99996	2	320512
matrix mul [29]	fully parallel, no comm during computation	8585229	8585228	16384	2	524288
convolution [29]	parallel → merge → parallel	448260	448259	65536	3	65536
dct [29]	fully parallel, no comm. during computation	2359298	2359298	262144	2	262244
merge sort [29]	parallel → merge → sequential	161233	157233	97668	2	39936
k-mean	parallel → merge → sequential (repeated)	1847765	1844981	36784	6	136192

TABLE III. BENCHMARK CHARACTERISTICS.

Name	Description	System	Latency
api-pci	mem copy using PCI-E	CPU+GPU, GMAC	33250+trans_rate
api-acq	acquire action	LRB	1000
api-tr	data transfer	LRB	7000
lib-pf	page fault	LRB	42000

TABLE IV. PARAMETERS OF MODELING COMMUNICATION OVERHEAD (TRANS_RATE IS 16GB/S IN PCI-E 2.0)

	Comp	UNI	PAS	DIS	ADSM
matrix mul	39	0	2	9	6
merge sort	112	0	2	6	4
dct	410	0	2	6	4
reduction	142	0	2	9	6
convolution	75	0	4	9	6
k-mean	332	0	6	6	4

TABLE V. THE NUMBER OF SOURCE LINES TO HANDLE DATA COMMUNICATION. COMP SHOWS THE NUMBER OF SOURCE LINES FOR COMPUTATIONS AND INITIAL DATA ALLOCATIONS.

number of source lines to indicate programmability.[5] We show the number of additional source lines required to handle explicit data communication and data handling operations in Table V. The result shows that the overhead increases in the following order: Unified $<$ partially shared $\leq$ ADSM $<$ disjoint memory space. Naturally, the unified memory space does not require any special APIs and the disjoint memory space requires the most additional source code lines. Since ADSM is also one type of the partially shared memory space, the overhead is somewhat similar. Although the number of source lines does not exactly indicate programmability, it still shows the trend that we expected. This result also verifies that the programmability of the partially shared memory space is between unified and disjoint memory spaces.

[5] We all know that the number of source lines does not necessarily mean programmability, but it is one of the few measurable metrics.

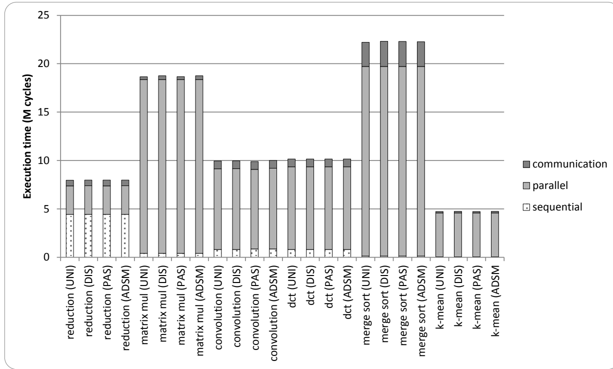

Fig. 7. Evaluation of memory address space design options with the ideal communication overhead.

D. Limitations of Our Evaluations

Although we discuss the locality management options, we could not evaluate the performance differences. The locality management option itself does not affect performance except for the additional instructions of `push` in the explicit management schemes. The overhead of software library implementations is roughly estimated since they are strongly dependent on actual implementations.

VI. Related Work

In this section, we discuss design space exploration work, especially work that targets heterogeneous computing.

A. Programming Model Comparisons

Several previous works on programming models for heterogeneous multiprocessors investigate the relationship between models and performance. They particularly target the IBM Cell architecture for this study. Schneider et al. evaluate three programming models for multiprocessors with explicitly managed memory hierarchies and analyze how the models manage parallelism and locality [32]. They develop benchmarks using each programming model and compare the codes themselves to show how implicit or explicit management of parallelism and locality affect programmer productivity. They also evaluate performance on the actual machines.

Ferrer et al. extend Schneider's work by considering memory bandwidth and more benchmarks [8]. They also classify programming models for the IBM Cell processors. Like Schneider's work, they show changes in the program in terms of lines, programs and tasks when the applications are parallelized. Since their work is only limited to the Cell architecture, they are able to only differentiate programming models. In contrast, we vary both programming models and hardware designs together.

B. Memory Models

Recently, several mechanisms have been proposed to reduce the cost of supporting coherence space [14], [33] and also the cost of consistency on GPUs [13], [12]. These approaches make a fully coherent and unified memory model more viable but they still have some constrains.

Kelm et al. propose the Rigel Architecture for a 1000-core accelerator [19]. Rigel targets accelerators not heterogeneous computing. However, the authors provide a detailed qualitative discussion on how they chose memory models for the architecture. Kelm et al. propose a hybrid memory model that uses both software and hardware coherence schemes [20]. Although the main focus of this work is on proposing a new cache coherence protocol, they analyze in great detail the trade-offs between hardware-managed and software-managed cache coherence. In our work, we explore various communication methods, address space design options, and locality control management schemes.

Leverich et al. compare coherent caches and a streaming memory model mainly for performance and power consumption [23]. Although their work has similarities to our work, they only evaluate two memory models: CPU and the streaming model. Furthermore, they have not considered design options of the shared memory space.

Patel and Hwu generalize accelerator architectures [30]. They show different hardware communication methods (i.e., integration methods) that are similar to what we have discussed in this paper. However, they only introduce these options to generalize accelerator architectures.

C. Heterogeneous Architecture Design Exploration

Lee et al. evaluate data-parallel accelerators and propose a new vector-thread architecture [22]. They explore microarchitectural design patterns to evaluate the trade-offs between programmability and implementation efficiency. Unlike our work, their work focuses only on core design configurations.

Mohanty et al. explore design spaces in heterogeneous embedded systems [26]. This study explores frequency, voltage, and memory sizes. The main focus of their work is on developing a fast design space exploration technique.

VII. Conclusions and Future Work

In this paper, we exploited the design space of heterogeneous computing memory systems. Although unified and coherent memory space is the ideal option for programmability, our study shows the trade-offs among different design options. The major conclusions from our comparisons are as follows:

1) The programmability of the partially shared memory spaces is between unified and disjoint memory spaces.
2) The design options of memory spaces and communication methods are mostly decoupled.
3) The partially shared address space allows the most number locality management options.

Our study indicates that the partially shared memory address space provides the most design options for architecture designs that can provide opportunities to optimize hardware and save power/energy. Especially, it provides various locality management options. Furthermore, it does not significantly increase the difficulty of programmability compared to the unified memory space as shown in Section 4.3. Our results also show that different memory space does not affect performance significantly. Hence, we conclude that partially shared memory space is the most promising design space option because of its many hardware design options and moderately good programmability.

In future work, we will develop metrics to measure the efficiency of design options to provide guidelines for future programming languages and future hardware system development.

Acknowledgments

We thank the anonymous reviewers for their valuable comments on our manuscript. This research was supported in part by the National Science Foundation under grant CCF 1054830. Jieun Lim was

partially supported by Brain Korea 21 Project. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect of NSF.

References

[1] "MacSim," http://code.google.com/p/macsim/.

[2] "UPC language specifications," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[3] AMD, "Fusion," http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx.

[4] J. Andrews and N. Baker, "Xbox 360 system architecture," *IEEE Micro*, vol. 26, pp. 25–37, 2006.

[5] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 48–57. [Online]. Available: http://doi.acm.org/10.1145/1122971.1122981

[6] H. Chen, Y. Gao, Z. Xiaocheng, S. Yan, P. Zhang, M. Rajagopalan, J. Fang, A. Mendelson, and B. Saha, "Unifed memory architecture (e.g., uma)," U.S. Patent application Number 20100118041, 2010.

[7] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188543

[8] R. Ferrer, P. Bellens, V. Beltran, M. Gonzalez, X. Martorell, R. M. Badia, E. Ayguade, J.-S. Yeom, S. Schneider, K. Koukos, M. Alvanos, D. S. Nikolopoulos, and A. Bilas, "Parallel programming models for heterogeneous multicore architectures," *IEEE Micro*, vol. 30, pp. 42–53, 2010.

[9] I. Gelado, J. H. Kelm, S. Ryoo, S. S. Lumetta, N. Navarro, and W.-m. W. Hwu, "CUBA: an architecture for efficient cpu/co-processor data communication," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 299–308. [Online]. Available: http://doi.acm.org/10.1145/1375527.1375571

[10] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10, 2010, pp. 347–358.

[11] D. Grewe and M. O'Boyle, "A static task partitioning approach for heterogeneous systems using opencl," in *Compiler Construction*, ser. Lecture Notes in Computer Science, J. Knoop, Ed. Springer Berlin / Heidelberg, 2011, vol. 6601, pp. 286–305.

[12] B. Hechtman, S. Che, D. Hower, Y. Tian, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood, "Quickrelease: A throughput-oriented approach to release consistency on gpus," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 189–200.

[13] B. A. Hechtman and D. J. Sorin, "Exploring memory consistency for massively-threaded throughput-oriented processors," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, Jun. 2013.

[14] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014.

[15] HP Labs, "CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model," http://www.hpl.hp.com/research/cacti/.

[16] IBM Corporation, "The Cell project at IBM Research," http://www.research.ibm.com/cell/, IBM.

[17] Intel, "Intel®Microarchitecture Sandy Bridge," http://www.intel.com/technology/architecture-silicon/2ndgen/index.htm.

[18] INTEL Corporation, "Intel®910gml/915g/915gm/915gms/915gv and 910gl express chipsets intel®dynamic video memory technology (dvmt) 3.0," http://www.intel.com/design/chipsets/applnots/30263.htm, INTEL Corporation, 2005.

[19] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 140–151.

[20] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: a hybrid memory model for accelerators," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[21] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han, "Comic: a coherent shared memory interface for cell be," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 303–314.

[22] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanovic, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011.

[23] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing memory systems for chip multiprocessors," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[24] J. Lim and H. Kim, "Design space exploration of memory model for heterogeneous computing," in *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, ser. MSPC '12, 2012.

[25] C. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[26] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation," in *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, 2002.

[27] NVIDIA, "Fermi: Nvidia's next generation cuda compute architecture," http://www.nvidia.com/fermi.

[28] ——, "Project denver," http://blogs.nvidia.com/2011/01/project-denver-processor-to-usher-in-new-era-of-computing/.

[29] *CUDA Programming Guide, V4.0*, NVIDIA Corporation.

[30] S. Patel and W.-m. W. Hwu, "Accelerator architectures," *Micro, IEEE*, vol. 28, no. 4, pp. 4 –12, july-aug. 2008.

[31] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson, "Programming model for a heterogeneous x86 platform," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 2009, pp. 431–440.

[32] S. Schneider, J.-S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos, "A comparison of programming models for multiprocessors with explicitly managed memory hierarchies," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009.

[33] I. Singh, A. Shriraman, W. Fung, M. O'Connor, and T. Aamodt, "Cache coherence for gpu architectures," *Micro, IEEE*, 2014.

[34] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, "Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.