RSVM: a Region-based Software Virtual Memory for GPU

Feng Ji*, Heshan Lin†, Xiaosong Ma*‡

* Department of Computer Science, North Carolina State University, Raleigh, NC, USA
  fji@ncsu.edu, ma@cs.ncsu.edu
† Department of Computer Science, Virginia Tech, Blacksburg, VA, USA
  hlin2@cs.vt.edu
‡ Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA

Abstract—While Graphics Processing Units (GPU) have gained much success in general purpose computing in recent years, their programming is still difficult, due to, particularly, explicitly managed GPU memory and manual CPU-GPU data transfer. Despite recent calls for managing GPU resources as first-class citizens in the operating system, a mature GPU memory management mechanism is still missing, which leads to reinventing the wheels in various GPU system software. Meanwhile, due to ever enlarging problem sizes, we urgently need a system-level mechanism for unified CPU-GPU memory management.

In this work, we present the design of Region-based Software Virtual Memory (RSVM), a software virtual memory running on both CPU and GPU in a distributed and cooperative way. In addition to automatic GPU memory management and GPU-CPU data transfer, RSVM offers two novel features: 1) GPU kernel-issued on-demand data fetching from the host into the GPU memory, and 2) intra-kernel transparent GPU memory swapping into the main memory. Our study reveals important insights on the challenges and opportunities of building unified virtual memory systems for heterogeneous computing. Experimental results on real GPU benchmarks demonstrate that, though it incurs a small overhead, RSVM can transparently scale GPU kernels to large problem sizes exceeding the device memory size limit; developers write the same code for different problem sizes, but still can optimize on data layout definition accordingly. Our evaluation also identifies missing GPU architecture features for better system software efficiency.

Index Terms—GPGPU, Heterogeneous System, GPU Memory Management.

I. INTRODUCTION

Graphics Processing Units (GPUs) have been widely adopted in modern computer systems. Equipped with tremendous parallelism, the throughput-oriented processor has been increasingly adopted for general purpose computing (GPGPU) over the past few years, especially for high performance computing applications. Although GPGPU programming models, e.g. CUDA [4] and OpenCL [5], are turning mature, programming on GPUs is still challenging. Especially, GPUs possess a discrete, off-chip device memory space not managed by the operating system’s virtual memory and connected through the bottleneck-prone PCIe bus. As the status quo, one must explicitly manage the GPU memory space and manually move data between it and the host memory. Optimizing data movement on the PCIe bus brings another factor of complexity. The manual programming overhead greatly increases programmers’ burden in writing GPU codes, limiting their productivity in application development.

Although the lack of CPU-GPU memory management has been recognized by the systems research community [14], [18], [26], a unified virtual memory solution remains unknown. Several recent projects to build compilers [17], [18], [25] and task scheduling runtimes [14], [26] share the need of managing CPU and GPU memory spaces.

However, as today’s GPU hardware provides no virtual memory support, long available in modern CPUs, existing compiler techniques rely on host-side static analysis to estimate a GPU task’s input and output data sets. Thus, they do not enable on-demand data fetching for the GPU task, i.e. transferring only the needed data across the PCIe bus at run time. For the same reason, existing runtime approaches either require non-transparent ways of specifying GPU kernels’ data demand upfront, e.g. using directed acyclic graphs [26], or achieve on-demand data fetching only from the host side [14].

We envision a unified virtual memory that hides the existence of discrete CPU and GPU memory domains. This unified virtual memory should support several important features missing in existing CPU-GPU memory management systems:

- It should present a unified address space to relieve programmers from manual CPU-GPU data movement.
- It should provide GPU memory management similar to the host-side paging mechanism.
- It should enable on-demand data fetching between the CPU and GPU memory spaces.
- It should provide a memory swapping mechanism for GPUs, giving a GPU task the illusion of a much larger device memory space.

In this paper, we present Region-based Software Virtual Memory, a CPU-GPU memory management approach, to explore the GPU virtual memory design space. We propose using regions as the basic data unit abstraction, which are defined by users and managed by RSVM for PCIe data movement and intra-kernel fine-grained GPU memory management. This design enables on-demand data fetching initiated from GPU kernels and a transparent swap mechanism backed by the host main memory. Meanwhile, an optimization space is still open.

- It should provide a memory swapping mechanism for GPUs, giving a GPU task the illusion of a much larger device memory space.
to users through adjusting their region definitions.

To summarize, this paper makes the following contributions:

- We describe the design of a software virtual memory on a GPU-powered heterogeneous system, and demonstrate its novel features desired for GPU programming.
- We propose using user-defined regions as memory managing abstractions for such systems, and show that such abstraction leaves users with a useful optimization space.
- We introduce several new GPU system software mechanisms to serve as RSVM’s building blocks, including distributed region table management, CPU assisted swapping, and a software region TLS in the GPU non-cache-coherent shared memory.
- We provide a proof-of-concept implementation of RSVM on systems with x86 CPUs and Nvidia CUDA discrete GPUs.
- We conduct an evaluation with real GPU benchmarks to evaluate RSVM and identify certain missing architecture features desired for efficient GPU system software.

II. BACKGROUND

GPUs have fundamental architectural difference compared with CPUs. In this section, we briefly introduce the architecture and GPU programming models, followed by a motivating example to demonstrate problems with current GPU memory management mechanisms.

![Fig. 1. Current heterogeneous system architecture](image)

<table>
<thead>
<tr>
<th>APP</th>
<th>Host Code</th>
<th>Device Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPGPU API</td>
<td>GPU Runtime</td>
<td></td>
</tr>
</tbody>
</table>

**GPU architecture:** A typical GPU consists of dozens of SIMD engines, called streaming multiprocessors (SM), and a high-latency, high-throughput memory, global memory (or device memory). In each SM, up to hundreds of simple SIMD cores are equipped with a large register file to support efficient hardware threading. A part of the per-SM L1 cache, called shared memory, can be used for fast data exchange across threads running on the SM.

GPUs are connected to the CPU (host) through a PCI-Express (PCIe) bus, whose bandwidth is orders-of-magnitude smaller than that of the GPU device memory. Thus, host-device data movement can easily become a bottleneck and hence deserves careful optimization.

**Current GPU system model:** In heterogeneous systems today, a GPU serves as a co-processor (as shown in Figure II). The CPU is responsible for managing GPU resources through a driver program. For GPU computing, GPGPU APIs are provided by the driver for commanding the device. An application running on the host side can asynchronously launch tasks on the GPU. The host can also submit a synchronized command, which blocks till all the preceding commands complete.

The GPU memory and main memory are connected by the PCIe bus, and no hardware-enforced cache coherence is provided between the two memory spaces today. A GPU can visit a part of the page-locked main memory by mapping the host-side memory address into its own MMU. CPUs cannot visit the GPU memory, but only CPUs can issue commands to move data across the two places. GPU programmers must manually manage the GPU memory and transfer data between the host and the device memory buffers.

**Problematic manual GPU memory management:** Figure 2 illustrates our target problem with a dense matrix multiplication implementation. The code consists of 4 steps: (1) GPU memory allocation, (2) input data transfer into the GPU memory, (3) kernel launch, and (4) result data transfer back to the host memory.

Several problems exist in the implementation. First, it assumes that all GPU buffers can be allocated, which holds true when the problem size is smaller enough and the available GPU memory resources are known beforehand. Second, manual PCIe data transfer brings coding overhead when performance optimization is adopted, as programmers must hard code for every different data transfer plan. Third, the static GPU memory allocation creates a larger memory working set for the GPU kernel than what the active threads actually need. As only a portion of all GPU threads are running at a given time, only the shaded sub-areas in the three buffers will be accessed with column-major thread scheduling.

III. REGION-BASED CPU-GPU MEMORY MANAGEMENT

In this section, we give an overview of our proposed virtual memory system for heterogeneous computing, called Region-based Software Virtual Memory (RSVM).

A. Region as Memory Management Unit

CPU-GPU memory management must first decide a basic unit for GPU memory management and PCIe data transfer.
Pages, typically used in CPU-based virtual memory, are often not the best option here. GPU threads are logically grouped into a grid of thread blocks, which are scheduled to run on a number of GPU SMs successively, following an order defined by the hardware scheduler. Therefore, using pages as the memory management unit may incure space inefficiency depending on hardware-specific thread block schedules. For example, consider the dense matrix multiplication program given in Figure 2. In cases where the hardware scheduler launches them in a column-major order, the working set for the active thread blocks is the shaded area in the three buffers.

Similarly, it is difficult to enforce a fixed, one-size-fits-all memory management unit for different applications. In this work, we propose the use of region [14], [19], a user-defined data block, as the basic memory management unit. Each region is either an arbitrarily sized, contiguous area in memory (1-D linear region), or a group of such contiguous areas of identical size and identical relative displacement (stride) in linear memory address (multi-dimension region). Currently, GPGPU APIs such as CUDA support linear, 2-D, and 3-D data layouts. In our proof-of-concept implementation and discussion in this paper, we focus on 1-D and 2-D cases due to their rather dominant adoption in GPU applications, while the RSVM design can accommodate regions of 3 or more dimensions.

When a region is created, one specifies its extents in each dimension, as well as the stride. In the case of 2-D regions, the extents in x and y dimensions are also called width and height. Each region is represented by a system-wide unique ID, which is generated at the creation time. A host-side maintained heap, sitting atop RSVM, creates a region with rsvm_malloc. The region can then be used through a map operation, which translates the region ID into an address that can be used as a standard pointer.

In the dense matrix multiplication example, let A, B, and C all be \(N \times N\) single-precision float matrices. A can be defined as a single 1-D region with \((\text{width}, \text{height}, \text{stride})\) of \((8N^2, 1, 0)\), B and C are multiple 2-D regions defined as \((8C, N, 8N)\), where \(C\) is the number of columns each thread block is assigned to compute (Figure 3). This way, the GPU working set for active thread blocks can be greatly reduced. The \texttt{unmap} interface marks the end of using a region in the GPU kernel code, so a region whose processing has completed can be removed from the working set. Note that multiple fine-grained regions are grouped into a region \texttt{collection} (rgn_coll) in the sample code, whose metadata \texttt{rgn_coll_meta} contains all the associated regions’ IDs. Section IV-A gives more details on RSVM APIs.

### B. Benefit of Region-based Approach

Adopting regions have multiple benefits. First, regions hide non-continuous memory domains’ management from developers: users need not care where a region is, but simply focus on creating and using it. Second, users can still optimize the granularity of memory access and PCIe data transfer, by adjusting the region definition taking into consideration of data distribution among GPU threads. Co-scheduled thread blocks can give runtime information regarding the working set to reduce GPU memory usage. Such optimization need to consider the tradeoff in region sizes according to individual application’s needs: large regions yield better PCIe transfer performance, while small regions reduces both wasted device memory space and unnecessary data transfer. Finally, assuming regions are correctly defined, the underlying system can leverage the working set information for system functions that cannot be done today, e.g. local memory domain buffering, on-demand data fetching, and transparent data swapping.

### Unified address space backed by regions

To some extent, the region ID space presents a unified address space across segmented CPU and GPU memory spaces. Such a unified address space is different from the current virtual address space, such as CUDA Unified Address Space (UVA) [4], which maps GPU memory into a distinctive segment in virtual memory address space and still requires explicit copying of data across CPU and GPU. In contrast, such copying is unnecessary for RSVM. Furthermore, pointer-based data structures must be marshalled across PCIe and reconstructed after.

Using this unified address space, Listing 1 shows a linked list prepared on the host side but consumed on the device. This feature is especially useful for CPU-GPU co-processing on shared data structures. Here an RSVM pointer consists of a region ID and an offset within the region. A host-side heap maintains a group of internal buffers, each of which, as a RSVM region, contains a configurable number of free elements for linked list nodes.

### IV. RSVM DESIGN

In this section, we describe the design of our prototype region-based CPU-GPU memory management system, focusing on addressing several unique challenges:

- **Parallelism**: traditional system design did not consider the huge parallelism possessed by GPUs.
Listing 1. A linked list in the unified address space

typedef struct {
    rsvm_id id;
    unsigned long long offset;
} rsvm_ptr_t; // a R SVM unified space pointer
typedef struct {
    type value;
    rsvm_ptr_t next;
} node_t; // a linked list node

// host code: host-side heap on R SVM,
// maintaining free elements in buf[n]
rsvm_malloc() {
    // ... when no free element can be found
    buf[n++] = rgn_create_cpu(sizeof_internal_buffer);
}

// host code: create a linked list
node_t *n = rgn_map_cpu(head.id, rsvm_ptr_t head = rsvm_malloc();
    op_writeonly, NULL, NULL) + head.offset;
    n->value = value;
    n->next = rsvm_malloc();
    // ... to add more nodes

// device code: traverse the linked list
rsvm_ptr_t traverse(rsvm_ptr_t head) {
    rsvm_ptr_t p = head;
    while (p != NULL) {
        rgn_map(p.id, op_readonly, &complete, &req) + p.offset;
        if (!complete) n = rgn_wait_map_gpu(
            p.id, req) + p.offset;
        else n = rgn_map_cpu(p.id);
        p = r;
    }
}

• Lack of synchronization tools: traditional software techniques, often utilizing synchronization methods like locks and semaphores, do not suit the throughput-oriented processor, where a large number of hardware threads are over-subscribed for latency hiding.

• Distributedness and asynchrony: memory access and management span both the GPU and the CPU sides, where computation proceed asynchronously.

• Obscurity and diversity: GPUs and the vendor drivers remain closed, with differences across vendors and generations. This requires the system assume the least on the GPU architecture and system support.

The architecture of RSVM is depicted in Figure 4, which shows its placement and internal software hierarchy. RSVM is built on top of the GPGPU APIs provided by either open-source or proprietary drivers. Below we describe the major components of RSVM.

A. Region API

GPU parallelism is created by the concurrent execution of multiple SIMD engines, each running warps of threads in lock steps. RSVM requires that threads within a warp invoke the same API function with the same parameters synchronously. Although warp specialization is possible, differentiating threads within a warp leads to SIMD lane divergence and is typically discouraged due to performance concern.

Table I summarizes the RSVM APIs. Most functions have both CPU and GPU versions, to be called in host/device codes. Among them, rgn_map does ID-to-pointer translation. Based on an op (readonly, readwrite, or writeonly) code, RSVM knows how users code access a region. It returns asynchronously even when a region’s data copying from the remote side is incomplete. In this case, the complete flag is reset and a request number is assigned. Instead of busy waiting, the program can perform other useful work before calling the blocking wait API, which returns upon the completion of data transfer. Note that a multi-dimensional region can be created only at the host side, as such more complex data structures are typically produced there and consumed by the GPUs. When it is mapped at the GPU, its stride might be modified for GPU memory’s alignment and returned in pitch. unmap, available as a device-side API only, marks the end of using a region on the GPU. Together with map, they form a memory access session in the GPU code, in order to let RSVM know how GPU code in the session accesses the region. RSVM relies on users to correctly pair map and unmap. This is unnecessary at the host side, as the OS virtual memory understands the working set of the process through paging.

The RSVM synchronization rsvm_sync, a host-side function, lets the asynchronous runtimes on the CPU and the GPU exchange their knowledge of updates to region states since the previous synchronization. Data updates across the PCIe bus are delayed to subsequent mapping operations. rsvm_sync can be invoked before a GPU kernel or after a synchronized GPU command, therefore regions will not experience a rsvm_sync event during a kernel function’s execution.

Table I also shows RSVM’s region collection-related utility functions. For example, rgn_coll_get_meta returns IDs of all the regions in the collection, in a local memory buffer.

B. Region States

A region’s state defines its current status, while the RSVM state transition protocol defines the state transition rules. RSVM uses an MSI-like protocol [15] as shown in Figure 5. Here, a modified region is one that has been modified since the last synchronization. A shared region has a clean local copy. An invalid region has no or obsolete data. In addition to the above three common states, the GPU side has two more, namely modifying and sharing. They have similar status with
the modified and shared states, but are identified as “currently in use” (by a pair of map and unmap operations). This allows the GPU-side swap and reclaim operations (see Section IV-E) to flush unmapped dirty data to host memory and release GPU-side buffers (along with region state changes).

C. Region Management

Beside a region’s state, RSVM internally stores its metadata, e.g. addresses in each memory domain, dimensions, in a region table, indexed by the ID. It has a copy on each side, with local changes merged at rsvm_sync operation. The region table, as well as the ID space, is statically partitioned into fixed-length segments. A segment is owned by either the CPU or the GPU at a given time, allowing only one side to create regions in its ID space. When a new region is to be created, RSVM finds an unused entry in a segment owned by the local side, which ensures a unique region ID without inquiring the other side. Obviously, if the region is deleted by the non-owner side, its region table entry will not be recycled by the owner till the next synchronization point.

A region manager sits on each side and serves user requests. It queries the region table for region states and a lower-level memory allocator to manage the local memory domain. Region managers communicate with each other for CPU-GPU cooperative tasks. When a region is created or mapped for the first time, the region manager creates a buffer for it in the local memory domain. If an invalid region is mapped, the region manager fetches its updatate copy into the local buffer. On the GPU, the region manager itself cannot initiate a PCIe data transfer and must communicate with the host side service through a GPU callback (Section V-A). Upon freeing a region, the manager returns the buffer to the memory allocator.

D. CPU-GPU region consistency

As the CPU and GPU work asynchronously, it is possible that one side is unaware of the modification to a shared region on the other. Not designed to invent new synchronization methods or consistency models, RSVM adopts a relaxed consistency model. This is based on the consideration that a strong consistency model would demand frequent synchronization across the PCIe bus, while most CPU-GPU co-processing applications have rather coarse-grained data/task distribution. As a result, RSVM reports the conflict when concurrent data modification is identified at runtime.

E. Enabling swap for GPU memory

unmap does not free memory resources immediately. RSVM devises a swap mechanism in the GPU region manager, using the host main memory as the backing store. Such swap operations can happen within a single large kernel, or after multiple kernel runs.

RSVM’s swap design considers two challenges. First, as GPU thread scheduling cannot be controlled, it is impossible to keep any daemon GPU threads which will wake up and are only responsible for swap. Second, region replacement policy requires online region access accounting and a victim algorithm to find candidate regions. Both can be substantial overhead with GPU’s massive parallelism.

As no GPU thread is dedicated for swapping, RSVM’s swap starting/stopping logic is embedded in map and unmap paths. However, instead of hijacking a GPU thread to complete a swap operation, the swap logic is designed asynchronous and re-entrant, and aided by a host-side assistance service. Swap and region reclamation are separated as two independent steps.

For region access accounting, RSVM keeps a Not-Frequent-Used (NFU) counter for each region in the GPU memory. It is incremented in each map operation. When the GPU memory availability is low, a GPU thread requests the host-side service thread to transfer the counters buffer—needed segments of the GPU’s region table—back into the host and start a traditional counter-based replacement algorithm, e.g. the aging algorithm RSVM currently employs. When the candidate regions are chosen, the host service starts PCIe data copies to swap those dirty ones out to the main memory. The host-side service does not modify the GPU-side region table, and does not block the signaling GPU threads.

As the swap is happening on the CPU, other GPUs thread may re-enter the swap logic in the GPU. Seeing an ongoing
request, they simply leave. When a thread finds the previous swap is done, it will see the swapped regions’ IDs returned from the last request, as well as the NFU sorted list. Among them those regions which have not been updated since last call will be set to the shared state, and become potential candidates for region reclamation.

Region reclamation happens as another independent operation of the GPU region manager. It simply reclaims the memory buffers for shared regions. It honors the latest NFU result, and changes the state of reclaimed regions to invalid.

F. A software TLB for region mapping on the GPU

Modern CPUs have a Translation Lookaside Buffers (TLB) cache for efficient converting virtual to physical memory addresses. As RSVM’s GPU runtime stores the region table in the device memory, it employs a software TLB mechanism using the fast on-chip shared memory inside each SM. As the shared memory is a thread-block-specific manually operated scratch-pad space, RSVM’s software TLB enforces coherence between the shared memory and the GPU device memory, and between shared memory spaces of two different SMs.

Coherence. Between the region table in the device memory and the cached portion in the shared memory, RSVM software TLB adopts a write-through policy.

To enforce coherence between two TLBs in different thread blocks, RSVM’s solution is that, when reading from a TLB entry, the threads distinguish safe and unsafe states. A region is safe, if its cached status is mapping or sharing states already. These states indicate that some other threads in this thread block are using it, which implies two facts: those threads have cached the region table entry, and that the TLB cached entry is updated, as the region is still in use, and no threads will try to modify its metadata. Otherwise, the thread proactively reads the region table entry to refresh the TLB entry and sends it to the safe state.

To coordinate the TLB entry state updating between thread warps, a reference counter is used in the shared memory. It is updated atomically in map and unmap operations. The first warp trying to use a region, i.e. the one successfully updates the counter value from 0 to 1, updates the global region entry (e.g. possibly changing the global region state) and prepares the TLB entry. Then the TLB entry is safe, and the rest threads need not touch the device memory until the last one, which will tryo update the global state.

TLB design. This software TLB implements a fully-associative policy, and enjoys a thread warp’s parallelism for look-up. Each entry has a status flag to indicate if the entry is mapped to a region or unused. A thread block-wise lock is used to protect changes to any entry’s status flag. After that, the entry’s consistency is maintained through atomically updating the reference counter and its state of the entry. When the reference count of a TLB entry is zero, it can be reused for caching another region entry. RSVM currently do not provide a TLB overflow evicting scheme, as that would violate the safeness mechanism above. As we leave the software TLB size as a configurable parameter, we assume that the user is able to estimate the maximum number of concurrently used regions by a thread block.

V. Implementation

This section will cover several important building blocks of RSVM.

A. GPU callback

GPU off-the-shelf systems today have no such a mechanism to let GPU code to invoke a host-side CPU function call. Yet RSVM’s GPU-side runtime need occasionally make such functions, e.g. when the GPU-side region manager tries to fetch the latest data for an invalid region. A software mechanism, called GPU callback, exists as a workaround, which has a CPU thread polling on a flag variable which is shared with a GPU [31].

In addition to the existing GPU callback mechanism, RSVM implements a new type of collective callbacks, for the case that many GPU threads ask for the same request. The GPU side runtime detects all concurrent GPU threads for the same callback, and sends only one single signal across. Upon completion, it lets one thread take care of finishing the task. Collective callbacks avoid the traffic surge on the PCIe bus and save the CPU from waisting repeated effort.

RSVM implements three types of GPU callbacks:

- **Handling a region fault**: non-collective, asynchronous, and parameterized callback. It asks for transmitting the data of an invalid region, and returns asynchronously a request number as well as a complete tag.
- **Getting a new region segment**: collective, synchronous, and non-parameterized callback. It asks for a new segment in the region table. It blocks all calling threads and sends a single signal to the host. It returns for all of them when a new segment is allocated to the GPU side.
- **Staring the swap**: collective, asynchronous, re-entrant, and non-parameterized callback. It asks the swapping service on the host side to fetch modified regions to the main memory. It returns asynchronously with a complete tag, which lets one thread to finish up on host-side completion.

B. Lower-level memory allocator on GPU

Similar to the host-side virtual memory system, a way of managing the underlying GPU physical memory resource is needed. RSVM relies on ScatterAlloc [30], a memory heap on the GPU. When RSVM is started, it is created with a predefined size, which limits the total GPU memory usage for a process. The host-side underlying memory management is reliant on a normal main memory heap.

C. Limitation of current implementation

RSVM is currently implemented as a user-level library, limiting its use in a single GPU application process. In the case of several host processes sharing the same GPU device, currently RSVM cannot dynamically adjust the GPU memory usage across processes as it lacks a global view. This can be extended by sharing the region table across multiple processes, utilizing shared memory mapping on the host side.
and cudaIPC shared memory functions on the device. RSVM is implemented currently for a single GPU device, but, in principle, can be extended to the multi-GPU scenario. Finally, users must launch their GPU kernels in a CUDA stream, which is required to overlap a GPU kernel with PCIe data transfers.

VI. Evaluation

In this section, we evaluate RSVM across a set of GPU benchmarks. Our testbed is an Intel x86 system with Xeon E5507 CPU, 6 GB main memory, and an Nvidia GTX480 Fermi GPU card (15 streaming multiprocessors and 1.5 GB device memory). The software is Ubuntu 10.04 LTS Server, with an x86_64 Linux kernel (2.6.32-33), and CUDA 5.0rc.

A. Benchmark

First, we ported a group of benchmarks from Nvidia CUDA SDK [4] and Rodinia benchmark suite [6] to RSVM. These workloads all have problem sizes that can fit into our GPU card’s device memory. Among them, matrix multiplication and BlackScholes are relatively computational intensive and HotSpot and BackProp are relatively memory intensive. When we define regions, we choose to take individual buffers as regions for HotSpot and BackProp as their buffers are small enough. For BlackScholes and MatrixMul, we define multiple fine-grained regions for each buffers. All experiments are repeated three times and taken average, and timing includes both computation and PCIe transfer time but not disk I/O or input generation time (the same for all experiments through this section).

Porting results are compared with their original native implementations (Table II). Three workloads, except for MatriMul, shows RSVM, as an extra software layer, introduces small overhead. However, for MatrixMul the slowdown is 21%. An examination reveals that RSVM brings a pressure on GPU register file, increasing the register usage count from 25 up to 60 (per GPU thread), as a few hundred lines of code from RSVM’s GPU device library, as well as ScatterAlloc, are compiled into the GPU kernel. Also note that with ScatterAlloc compiled in, register count is already 32. Required by GPU hardware threading, register files are statically partitioned to each GPU thread. The increase in register usage decreases the streaming multiprocessor occupancy, i.e. the simultaneous active threads per SM is decreased from 1024 in the native version to 512. MatrixMul, a computational intensive workload, has no other resource bottleneck on GPUs as for other three workloads. Thus a decrease in occupancy shows rather large slowdown.

\[
\begin{array}{|c|c|c|}
\hline
\text{Workload} & \text{Problem size} & \text{Slowdown} \\
\hline
\text{MatrixMultiplication} & 8192x8192x8192 (768 MB) & 21% \\
\text{BlackScholes} & 33554432 (64 MB): 12 iterations & 5% \\
\text{HotSpot} & 1024x1024x1024 (12MB): 16 iterations & 1% \\
\text{BackProp} & 262144 (K1:19.93MB/K2:36.71MB) & 5% \\
\hline
\end{array}
\]

Table II. RSVM vs native in benchmarks that fit in the device memory (BackProp has two kernels)

Fig. 6. RSVM vs. native/manual matrix multiplication with increasing size (sum of input and output matrices)

B. Matrix multiplication with increasing problem size

In this experiment, we use matrix multiplication to evaluate RSVM when problem sizes exceeds the GPU device memory limit. Although MatixMul shows slowdown on RSVM in the previous example, we still choose it, because it is typically used to evaluate hardware computational power (GFLOPS).

As the problem size increases beyond the GPU memory limit, the native version will not run. For performance comparison, we implement a manual version. It partitions matrix B and C into multiple column-wise sub-matrices, and creates buffers in GPU memory that can store matrix A and a partition from matrix B and C. It takes iterations to upload each partition to the GPU memory, run the computation kernel, and download the result partition to C.

The RSVM version uses the same implementation with same parameter for all input sizes. As this matrix multiplication turns out to have a column-major scheduling for launching thread blocks, we choose the region definition as in Figure 3. Matrix A is created as a single region. Matrix B and C are created as 16 floats wide 2-d regions, with their heights and strides as the height and the width of Matrix B and C, respectively.

The result is shown in Figure 6. First, note that the GPU memory size is 1.5 GB, but, in RSVM, we can use only 1280 MB at most for the managed GPU memory. Although our RSVM library uses only a few dozen MB internally, it turns out that the GPU reserves some memory for its own use, and CUDA’s GPU cudaMalloc allocator has probably fragmentation issues, which are only shown in creating various different GPU buffers inside RSVM. Due to the cubic computation complexity, the manual version attains a saturated GFLOPS. RSVM version, despite occupancy problem, almost steadily reaches 70% of the saturated throughput, although the swap mechanism starts to kick in at 1200 MB. After all, RSVM survives the large problem size, requiring no code change, and its swap mechanism shows less than 10% overhead.

Alternatively, we test a wrongful region definition, by defining multiple linear regions for matrix A and C and a single region for matrix B. While it can run at 768 MB, with an 25%
overhead below the correct RSVM version, thrashing occurs at 1200 MB. This shows that it is better to leave choices of region definition to application developers.

C. Breadth-first search

The third experiment is a breadth-first search (BFS) graph algorithm. Graph algorithms are known for dynamic memory access patterns, which depend on input graphs and are hard to predict before running. Therefore, it is hard for programmers to find an optimal schedule as with MatrixMul. Meanwhile, BFS is a memory intensive workload, which we use to evaluate RSVM in contrast to computation-intensive MatrixMul.

The BFS native implementation is from SHOC benchmark [10]. It implements a warp-centric algorithm [16], by parallelizing vertices among different warps, and the edges adjacent to the same vertex among the threads of that warp. The graph is prepared in an adjacent list format with an array of edges and an index array for each vertex into the edge array. To port it to RSVM, we cannot know how threads are accessing the data upfront, and thus choose to define two arrays each as a collection of same-sized linear regions—we leave the region size as adjustable parameters. Another cost array for each vertex to record its distance from the source is created as a single large region.

The input graphs we prepared are from the 10th DIMACS Challenge [1] except for rmat and rmat-g500, which are generated from GTgraph [3]. Of a graph with \( n \) vertices and \( m \) edges, the edge factor \( \frac{m}{n} \) gives the information of the graph’s structure. It is also related to BFS iterations, or depth. In Table III, the upper 5 graphs (“small”) have problem sizes that can fit into our GPU memory limit; the lower 4 graphs (“large”) are chosen to exceed the limit of 1280 MB.

The algorithm’s performance are measured in traversed edges per second, or TEPS.

**Table III. Benchmark graphs (N and M in 10^6). Small (upper, fit GPU Memory Limit) and Large (lower, exceed GPU Memory Limit). “Kernel”: Launching Configuration as “Number of Blocks/Block Size”.

<table>
<thead>
<tr>
<th>Name</th>
<th>n</th>
<th>m</th>
<th>( \frac{m}{n} )</th>
<th>depth</th>
<th>kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>nlpkkt200</td>
<td>16.2</td>
<td>432.0</td>
<td>26.6</td>
<td>203</td>
<td>248/256</td>
</tr>
<tr>
<td>arabic-2005</td>
<td>22.7</td>
<td>640.0</td>
<td>28.1</td>
<td>98</td>
<td>248/256</td>
</tr>
<tr>
<td>rmat</td>
<td>4.19</td>
<td>536.9</td>
<td>128</td>
<td>5</td>
<td>62/256</td>
</tr>
<tr>
<td>rmat-g500</td>
<td>4.19</td>
<td>536.9</td>
<td>128</td>
<td>5</td>
<td>62/256</td>
</tr>
<tr>
<td>nlpkkt200</td>
<td>16.2</td>
<td>432.0</td>
<td>26.6</td>
<td>203</td>
<td>248/256</td>
</tr>
<tr>
<td>arabic-2005</td>
<td>22.7</td>
<td>640.0</td>
<td>28.1</td>
<td>98</td>
<td>248/256</td>
</tr>
<tr>
<td>rmat</td>
<td>4.19</td>
<td>536.9</td>
<td>128</td>
<td>5</td>
<td>62/256</td>
</tr>
<tr>
<td>rmat-g500</td>
<td>4.19</td>
<td>536.9</td>
<td>128</td>
<td>5</td>
<td>62/256</td>
</tr>
</tbody>
</table>

Fig. 7. RSVM vs. native BFS on small graphs

Fig. 8. RSVM vs. manual/UVA BFS on large graphs

For the last one, a GRAPH500 [2] input graph, RSVM shows 4% improvement over the native implementation, due to amortized overhead and overlapping of GPU kernel and PCIe data communication.

**Large graphs.** For large graphs, BFS native version will not run. Therefore we implement two normal ways to deal with the case. 1) **Manual:** it partitions the graph into multiple parts, each fitting into the GPU memory. Then, it take turns to run BFS on each part for each BFS iteration. Data swapping is manually done across the host side buffers with the GPU memory partition buffers for each turn. 2) **UVA:** this method leaves the buffers on the host, and maps them into GPU’s address space so that GPU threads can access them directly, but through the PCIe bus. One should notice that optimizations can be taken onto them, e.g. double buffering for the manual approach to overlap computation with data communication. However, two above methods are fundamentally different, with rather distinctive code bases to maintain.

Figure 8 shows their performance methods compared with RSVM, across four different graphs with similar sizes (1.7 GB - 2.5 GB). Between manual and UVA approaches one wins over the other depending on graph structures. For graphs with small edge factors and long depths, the manual approach performs worse, as excessive data is constantly swapping into and
out of GPU each turn, with very small portion used actually, while the UVA version is good at accessing only the needed data over the PCIe bus. For graphs with large edge factors and low depths, the manual approach performs better as it transmits a partition at a time into the GPU memory, benefiting from PCIe efficiency and local data accesses, whereas in the UVA version simultaneously running threads access adjacency list data over the PCIe bus, making it a bottleneck.

RSVM enjoys the benefit of both, as region sizes can be adjusted, and are fetched into GPU memory on demand. The results show that RSVM meets the better performance between manual and UVA methods, and even surpasses them for two R-mat generated low-depth graphs.

D. Discussion

As shown in the experiment of MatrixMul, added register file pressure due to the RSVM library can hurt the performance very much. Static register file partitioning strategy counts the same number of registers to all threads. However, the RSVM library code is executed in only a small number of GPU threads at a time. Dynamically sharing registers between threads will be able to reduce such register file pressure.

Lack of a privilege mode on GPU leaves RSVM device library data structures under-protected. Misbehaving GPU threads in the user code can pollute RSVM’s data structure. This has led to debugging difficulty in our experience, and is also a security issue.

VII. RELATED WORK

Program non-continuous memory hierarchy Accelerators such as the Cell BE processors [9], GPUs, and Larrabee processors [28] have gained increasing popularity among high performance computing applications. However, they present a non-continuous memory hierarchy, requiring manual data movement. Plenty of work has been done to facilitate transparent data movement for specialized programming models. Such as MapReduce and directed acyclic graph based algorithms [8], [11], [13], [22], [23]. Transparent software caching techniques on accelerators have been proposed for CELL BE and Larrabee [12], [27], [32], by utilizing their virtual memory support, which is missing on today’s GPUs [24]. RSVM demonstrates the potential of software virtual memory without requiring hardware or OS support on GPUs.

Compiler assisted CPU-GPU communication Compiler techniques have been developed to attain automatic CPU-GPU communications [14], [17], [18], [25], without requiring virtual memory exception support on the GPU. Static program analysis techniques are applied to GPU kernel code, to estimate GPU kernels’ input and output data set, and a host-side runtime mechanism is employed to check states and initiate data movement. However, the static analysis on GPU code is ineffective in dealing with dynamic data movement requirement, when data access is decided dynamically or when the working set exceeds GPU physical memory size limit. RSVM resolves these issues through a GPU-side runtime and callbacks. The dynamic memory footprint revealed by RSVM relieves compilers from conservative estimation of GPU data demand. Meanwhile, compilers can help RSVM in identifying regions, by analyzing locality in the GPU kernel code.

OS support for GPGPU Though GPUs have been used in modern computer systems for over a decade, system designers have just noticed the limited OS authority over the GPU resources, as they are turning into compute devices. Previously delegated to GPU device drivers, scheduling, data flow, and QoS management of GPU tasks can be incorporated into the OS kernel space, e.g., GDev [20], possibly through new OS abstractions, e.g., PTask [26]. While they are more concerned with the synergy of current GPU programming models with the OS, RSVM is novel in extending system-level management into GPU kernels, to achieve transparent swapping and fine-grained on-demand data movement.

Distributed shared memory RSVM is inspired by distributed shared memory systems in the past [21] and particularly by CRL [19], an all-software distributed shared memory. Similar ideas were first introduced in ADSM [14], which abstracts a pair of user-managed host-side and device-side buffers. RSVM, however, adopts a different design and provides new features such as GPU-side on-demand data fetching and transparent swap. Very recently, GPUs [29] provides POSIX-like file I/O APIs for GPU kernels. It uses a device-side runtime to implement a buffer cache on the GPU. Despite the resemblance in design, RSVM’s memory-level interface is more flexible in programming CPU-GPU data sharing and co-processing. Meanwhile, GPUs’ functionality can be implemented on top of RSVM, by binding a region’s host-side address with memory-mapped files.

GPU virtual memory architectural evolution Today’s GPUs lack hardware virtual memory exception support, prompting RSVM to adopt a software approach. At the same time, new architecture such as Heterogeneous System Architecture (HSA) [7] is emerging for on-chip, integrated GPUs. This avoids the manual memory management problem by employing a single memory for both CPUs and GPUs. While its programming benefit is appealing, the current practice of deploying different memory types for CPUs and GPUs will likely offer performance advantage to GPGPU workloads.

VIII. CONCLUSIONS

GPU virtual memory provides both ease of programming and performance benefits. As of today, due to the lack of architectural support, there are no virtual memory systems that abstract/unify discrete host and GPU memory spaces. In this paper, we present RSVM, our exploratory software virtual memory system and demonstrate its convenience in program- ming as well as performance improvement. Meanwhile, RSVM is shown to incur rather significant overhead in certain use cases, revealing the need for additional architectural features for future-generation GPUs to benefit more from software virtual memory management.

ACKNOWLEDGMENT

We thank the anonymous reviewers and the Synergy group at Virginia Tech for their valuable feedback on this project. This work has been sponsored in part by an NSF project. This work has been sponsored in part by an NSF program. This work has been sponsored in part by an NSF program.
Awards CNS-0546301 (CAREER) and CNS-0915861, plus Xiaosong Ma’s joint appointment between NCSU and ORNL.

REFERENCES


