ABSTRACT

Heterogeneous systems equipped with traditional processors (CPUs) and graphics processing units (GPUs) have enabled processing large data sets. With new programming models, such as OpenCL and CUDA, programmers are encouraged to offload data parallel workloads to GPUs as much as possible in order to fully utilize the available resources. Unfortunately, offloading work is strictly limited by the size of the physical memory on a specific GPU. In this paper, we present Virtual Address Space for Throughput processors (VAST), an automatic GPU memory management system that provides an OpenCL program with the illusion of a virtual memory space. Based on the available physical memory on the target GPU, VAST does the following: automatically partitions the data parallel workload into chunks; efficiently extracts the precise working set required for the divided workload; rearranges the working set in contiguous memory space; and, transforms the kernel to operate on the reorganized working set. With VAST, the programmer is responsible for developing a data parallel kernel in OpenCL without concern for physical memory space limitations of individual GPUs. VAST transparently handles code generation dealing with the constraints of the actual physical memory and improves the re-targetability of the OpenCL with moderate overhead. Experiments demonstrate that a real GPU, NVIDIA GTX 760 with 2 GB of memory, can compute any size of data without program changes achieving 2.6x speedup over CPU execution, which is a realistic alternative for large data computation.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors—Run-time environments; Code generation

General Terms
Design, Experimentation, Performance

Keywords
Virtual memory, GPU, Optimization

1. INTRODUCTION

Graphics processing units (GPUs) have emerged as the computational workhorse of throughput oriented applications because of their low cost, high performance, and wide availability [11]. Modern GPUs achieve several TFLOPS of peak performance while costing a few hundred dollars. The advent of new programming models, such as OpenCL [7] and CUDA [5], makes it possible to utilize GPUs for processing massive data in parallel for general purpose applications. By leveraging these programming models, programmers can develop data parallel kernels for GPUs that achieve speedups of 4-100x over traditional processors (CPUs) [24].

Unfortunately, discrete GPUs have a critical limitation: the entire data to process must reside in GPU memory before execution. When the data size exceeds the physical memory of the GPU, the user must fall back to the CPU which supports nearly arbitrary data sizes through virtual addressing. Smart programmers can overcome this restriction by applying a divide-and-conquer approach. In this case, the programmer explicitly divides the workload into smaller chunks and executes a series of kernels each processing a subset of the data. This approach is not a panacea, however. Even in the simple case where the kernel operates on contiguous data chunks, explicit data management is tedious, requiring explicit buffer allocations and data transfer to/from the host. For non-contiguous data, divide-and-conquer is more complex. Lastly, partitioning decisions change across GPUs with differing amounts of physical memory.

A natural question is why do GPUs not support virtual addressing to eliminate this problem as CPUs have done for many decades [26, 10]. Through a combination of hardware (e.g., translation lookaside buffers) and operating system support (e.g., page tables), CPUs provide the appearance of a nearly infinite address space to facilitate processing large data sets. However, virtualizing the address space of discrete GPUs is difficult for the following reasons:

- Discrete GPUs have separate memories that use different address spaces, and each transaction between the host and GPU is expensive.

- GPUs do not interact with the operating systems for memory management, but instead directly access their physical memory.

- Execution on GPUs is non-preemptive, therefore all data must be present in the physical memory before execution.

The combination of these factors makes it difficult to execute kernels on GPUs whose total memory footprint exceeds the physical memory size on the GPU without programmer intervention. As the size of data to process keeps increasing, this limitation will become more significant.
To tackle these challenges, we present Virtual Address Space for Throughput processors (VAST), a run-time software system that provides the programmer with the illusion of a virtual memory space for commodity OpenCL devices. VAST transparently divides and transfers working sets based on the available physical memory space on the target GPU and automatically merges partial outputs. To virtualize the memory space, VAST adopts a look-ahead page table (LPT), a new type of page table that contains a list of virtual pages that will be accessed by specific ranges of the OpenCL workload. LPT differs from conventional page tables used in operating systems in that the LPT is filled up before the pages are actually accessed. With LPT, VAST decomposes the working set into individual page frames for execution of the partial workload. Page frames are packed into a contiguous buffer (frame buffer) that will reside in the GPU's physical memory and LPT represents the mapping of an OpenCL buffer from the CPU's virtual space into the GPU's physical space (the frame buffer). Instead of transferring the entire data to the GPU, VAST transfers the LPT and frame buffer for each partial workload. At the same time, VAST transforms the kernel to access memory through the LPT (e.g., software address translation).

The VAST runtime system significantly improves GPU portability as it can utilize any GPU for larger sized workloads. The challenges of VAST are four fold: dividing an arbitrary workload based on the available physical memory size, efficiently generating the LPT and frame buffer, transforming the kernel to use the relocated and packed data, and avoiding replicated transfers due to reuse of data across partial workloads. To address these issues, this paper makes following contributions:

- A code transformation methodology that quickly inspects memory access locations of an OpenCL kernel in order to generate LPTs.
- A novel technique that partitions an OpenCL workload into partial workloads based on the physical memory constraints of a GPU and packs the corresponding data into a frame buffer using LPTs.
- Kernel transformation techniques to access data out of the frame buffer using LPTs.
- A technique to avoid replicated data transfers to the GPU.
- A comprehensive performance evaluation of VAST on real hardware consisting of an Intel Core i7 CPU and an NVIDIA GTX 760 GPU.

The rest of the paper is organized as follows. Section 2 discusses the OpenCL execution model and opportunities for virtualizing GPU memory space. Section 3 explains the overview of VAST, and then the implementation is discussed in Section 4. The experimental results of using VAST for various OpenCL applications are presented in Section 5. Section 6 discusses the related work in this area. And finally, we conclude in Section 7.

2. BACKGROUND AND OPPORTUNITIES

This section briefly describes the OpenCL execution model and discusses memory consistency of OpenCL to understand the semantics of executing partial workloads of an OpenCL kernel.

2.1 OpenCL Execution Model

In OpenCL, the basic unit of execution is a single work-item which corresponds to a thread. A group of work-items executing the same code are weaved together to form a work-group. These work-groups are combined to form a unit of execution called NDRange, N-Dimensional Range, where each NDRange is scheduled by a command queue. For the execution, the OpenCL program assumes that the underlying devices consist of a number of compute units (CUs) which are further split into processing elements (PEs). When executing a kernel, work-groups are mapped to CUs, and work-items are assigned to PEs. In order to launch a kernel, a programmer must define the number of work-groups in NDRange, and the number of work-items in a work-group. In real hardware, since the number of actual cores are limited, CUs and PEs are virtualized by the hardware scheduler or OpenCL drivers.

2.2 Memory Consistency and Partial Execution

OpenCL uses a relaxed memory consistency model for local memory within a work-group and for global memory within a kernel’s workspace, NDRange. Each work-item (thread) in the same work-group sees the same view of local memory only at a synchronization point where a barrier appears. Similarly, every work-group in the same kernel is guaranteed to see the same view of the global memory only at the end of kernel execution, which is another synchronization point. This means that the ordering of execution does not have to be guaranteed across work-groups in a kernel, but guaranteed only across synchronization points [20].

Based on this memory consistency model, an OpenCL kernel can be executed in parallel at a work-group granularity without concern of the execution order. If a kernel executes a subset of work-groups instead of the entire NDRange, the result at the end of kernel execution would be incomplete. However, if the rest of the work-groups are later executed, it would correctly complete. This feature enables splitting workloads without concern for write-conflicts among work-groups [20]. By simply assigning a subset of work-groups for execution exclusively, partial results would appear interleaved in the memory space. Correct final results are completed when every work-group finishes the execution.

Through the code transformation, the host program can control the number of executed work-groups as shown in Figure 1. By inserting checking code at the beginning of the kernel, every work-
3.1 VAST System Execution Flow

VAST compiles an OpenCL kernel into intermediate representations (IRs), and then generates a GPU binary (PTX) for the Inspector Kernel. After a compilation request, the application will request the kernel launch. On this request, VAST first launches the Inspector Kernel that only inspects usage of global arguments (arguments with __global keyword) and fills up the Page Accessed Sets (PASs). PAS contains boolean values that represent whether a work-group has accessed each page. Note that each work-group has its own PAS for each global argument of the kernel, and the size of PAS is fixed as \( \text{PageSize} \cdot \text{NumGlobalArg} \) where \( \text{PageSize} \) is actual OpenCL buffer allocation size from the application. For example, 4GB of OpenCL buffer will require 1MB of PAS with 4KB pages. In order to reduce the number of PASs, one PAS can be shared among several work-groups. In this case, one PAS represents the pages accessed by a subset of work-groups.

Next, VAST transfers PASs from the GPU to the host, and then the host fills up the LPT and frame buffers using the PASs until the size of frame buffer reaches the available GPU memory size. At this point, VAST knows how many work-groups should be executed as it has generated the LPT and frame buffer for the specific range of work-groups. In order to execute specific range of work-groups of the kernel, the Paged Access Kernel also takes additional parameters for the ranges of work-groups to execute as discussed in Section 2.2. Once VAST finishes the transfer, it launches the Paged Access Kernel, which accesses the memory through the LPT.

During execution, the frame buffer in the device memory will be updated, and after the execution, VAST will transfer them back to the host memory space. Finally, VAST recovers arrays using the LPT and modified frame buffer, and repeats the steps from LPT generation to recovery until all work-groups finish their execution.

Figure 2: The VAST system located between applications and OpenCL library. VAST takes an OpenCL kernel and transforms it into the inspector kernel and the paged access kernel. At kernel launch, the GPU generates PASs (\( \text{PASgen} \)) by launching the inspector kernel, then transfers them to the host to create LPT and frame buffer (\( \text{LPTgen} \)). Next, LPT and frame buffer are transferred to the GPU in order to execute the paged access kernel.
3.2 VAST Execution Timeline

Figure 3 visualizes the timeline of VAST execution. As shown, the cost of generating LPTs and frame buffers can be hidden by overlapping them with data transfer and kernel execution. Thus, the only exposed cost is PAS generation and the first LPT generation. Mind that VAST allocates LPTs and frame buffers twice on the host for double buffering. With double buffering, the next LPT generation can proceed right after the previous LPT generation, as shown in Figure 3.

One important point from Figure 3 is that transferring the second working set to the GPU (marked as *) starts after retrieving the first partial result back to the host, not after the kernel execution. The main reason is that some pages written in the first partial execution can also be written by the work-groups in the second partial execution. Similar to data forwarding in a CPU pipeline datapath, VAST forwards only written pages from the previous execution to the next execution. Details of forwarding shared pages will be discussed in Section 4.4.

Another feature of VAST is that it avoids duplicated data transfer. To illustrate, if page frames of a global argument during the partial execution are identical to that of the next execution, VAST skips LPT and frame buffer generation as well as buffer transfer for the next execution.

4. IMPLEMENTATION

This section discusses the implementation of VAST system including design of PAS, kernel transformations, and how to create LPT and frame buffers using PASs. Also, the ways to handle shared pages and to avoid duplicated data transfers are discussed.

4.1 The Design of Page Accessed Set

As described in Section 3.1, the first step for VAST is launching the inspector kernel on the GPU in order to generate Page Accessed Sets (PASs). PAS represents a list of pages accessed by a set of work-groups during execution. Therefore, each entry of PAS contains a boolean value as shown in Figure 4. Later, these PASs will be used on the host to generate Look-ahead Page Table (LPT), which contains pointers to the page frames.

The inspector kernel is able to execute on the GPU without transferring the working set because most OpenCL kernels use a combination of work-item index and scalar variables as the index of global array access. Upon this property, VAST passes only scalar and __constant arguments to the inspector kernel along with the NDRange information (size of work-groups and work-items) for PAS generation. If global array access depends on the value of a global argument, (e.g. indirect memory access), the inspector code is leveraged on the host, and the host generates PASs directly.

As shown in Figure 4, VAST makes use of single level paging because it minimizes both PAS generation time for the inspector kernel and address translation time for the paged access kernel. As a result, with 4KB pages, 4 GB of data can be fit into 1 MB of PAS if each entry uses 1 byte to store a boolean value. However, if PASs are maintained per work-group per global argument, the size of overall PASs can become significant if the kernel is launched with a large number of work-groups. For this reason, VAST makes a set of consecutive work-groups to share one PAS depending on the number of work-groups. The number of work-groups per PAS is determined statically using the equation below.

\[
\text{WORKGROUP}_\text{PER}_\text{PAS} = \text{Ceil}(\frac{\text{MEMSIZE}}{\text{AllocSize}}) \quad (1)
\]

The assumption behind this equation is that one work-group accesses the space of \(\text{MEMSIZE} \div \text{AllocSize}\). With this assumption, VAST computes the number of work-groups not to exceed \(\text{MEMSIZE} \div \text{AllocSize}\) accesses. This is a rough estimation, and if it appears that the number is too small or large, VAST adjust the number dynamically.

When several work-groups try to modify the shared PAS, atomic operation is not necessary because according to NVIDIA’s programming guide [5], if a non-atomic instruction executed by more than one thread writes to the same location in global memory, only one thread performs a write and which thread does it is undefined. Thus, it is safe to share one PAS among several work-groups and work-items because every work-item will try to write the same boolean value (TRUE).

4.2 OpenCL Kernel Transformation

As shown in Figure 2, VAST transforms each OpenCL kernel into an inspector kernel and paged access kernel. Both kernel transformations only focus on usages of global arguments, but the difference is that the inspector kernel replace entire memory operations with the new stores, while the paged access kernel replaces only the bases and offsets of memory operations as shown in Figure 5.
Once the kernel arguments are setup, VAST performs several steps to transform the kernel as illustrated in Figure 5(a). First, it depends on the actual OpenCL buffer allocation size from the respective global arguments. Besides, the arguments for PAS to the kernel, each of which corresponds to arguments for PASs to the kernel, each of which corresponds to arguments for PASs to the kernel, each of which corresponds to arguments for PASs to the kernel. Because each work-group finds out their own PAS base by offsetting the PAS argument by $flat_work_group_idx \times PAS\_SIZE$ as shown in Figure 4. Note that the size of PAS differs by global argument because it depends on the actual OpenCL buffer allocation size from the application.

In detail, the first step for the inspector kernel transformation is to inline function calls in the kernel in order to avoid expensive inter-procedural analysis. In general, every function call can be inlined since the OpenCL programming model prohibits recursive calls as it adopts SIMT execution model that does not allow threads to execute different instructions at a time [20]. Next, VAST adds arguments for PASs to the kernel, each of which corresponds to the respective global arguments. Besides, the arguments for each PAS’s size are added to the inspector kernel because each work-group finds out their own PAS base by offsetting the PAS argument by $flat_work_group_idx \times PAS\_SIZE$ as shown in Figure 4. Note that the size of PAS differs by global argument because it depends on the actual OpenCL buffer allocation size from the application.

Once the kernel arguments are setup, VAST performs several steps to transform the kernel as illustrated in Figure 5(a). First, VAST finds out load (LD) or store (ST) instruction that accesses memory parallelism. Once the nodes for the frame pointer and the frame buffer as shown in Figure 5(b). As shown in the figure, an additional LOAD is used for querying the target frame index. The LOAD address of LPT is likely to be the same among work-items, because OpenCL programmers are encouraged to write the kernel to access the memory coalesced between work-items to fully utilize memory parallelism. Once the nodes for the frame pointer and the frame offset are added, VAST replaces the original base and offset operands of memory access with the frame pointer and the frame buffer as shown in Figure 6. As shown in the figure, an additional LOAD is used for querying the target frame index. The LOAD address of LPT is likely to be the same among work-items, because OpenCL programmers are encouraged to write the kernel to access the memory coalesced between work-items to fully utilize memory parallelism. Once the nodes for the frame pointer and the frame offset are added, VAST replaces the original base and offset operands of memory access with the frame pointer and the frame offset respectively as shown in Figure 5(b).

**Pointer Aliasing** is also handled in VAST during kernel transformations. As OpenCL kernels can use a local pointer variable, aliased global pointers also must be considered for both the inspector kernel and the paged access kernel. An important property of an OpenCL program is that the __global keyword implicitly in-
Algorithm 1 PAS Reduction

1: \( k = 1 \)
2: RangeFrom = 1
3: ReducedPAS[1..NumArgs][1..PAS_SIZE] = 0
4: MemFullPAS[1..1][1..NumArgs][1..PAS_SIZE] = 0
5: for \( i = 1 \) to \( P \_\text{CNT} \) do
6: TotalMemUsed = 0
7: for \( j = 1 \) to \( NUM\_\text{ARGS} \) do
8: MemFullPAS[k][j] = ReducedPAS[j]
9: ReducedPAS[j] = OR\_REDUCE(ReducedPAS[j], PAS[j][i])
10: TempUsed = SUM\_REDUCE(ReducedPAS[j] \times PAGE\_SIZE)
11: TotalMemUsed = TotalMemUsed + TempUsed
12: end for
13: if TotalMemUsed > MEM\_SIZE then
14: \( i = i + 1 \) \( \triangleright \) Roll-back one step
15: \( k = k + 1 \)
16: StoreRange(RangeFrom, i) \( \triangleright \) Store partial execution range
17: RangeFrom = \( i + 1 \)
18: ReducedPAS[1][1..NumArgs][1..PAS\_SIZE] = 0 \( \triangleright \) Reset
19: end if
20: end for
21: StoreRange(RangeFrom, PAS\_CNT) \( \triangleright \) Store the last range
22: return MemFullPAS[1..k] \( \triangleright \) return reduced PASs

cludes restrict keyword, thus global arguments will not alias each other (No-Alias). In other words, any pointer variable that will alias to a global argument must be declared with the \_global keyword. As a results, the aliasing of all global pointers can be determined as either No-Alias or Must-Alias through basic alias analysis. With this alias result, VAST can keep track of aliased pointers by offsetting the base. If an aliased global pointer uses different type (typcast), VAST uses different \textit{SHIFT\_AMT} for the aliased pointer during transformation.

4.3 Look-ahead Page Table Generation

Once PASs are generated by launching the inspector kernel, those are transferred to the host in order to generate Look-ahead Page Table (LPT) and frame buffer. As shown in Figure 4 and 6, LPT differs from PAS in that the entry of LPT contains a frame pointer (4 bytes), while PAS contains a boolean variable (1 byte). Moreover, LPT stands for one global argument, while PAS is maintained per global argument per work-groups as discussed in previous sections. LPT can be produced multiple times for one global argument because one LPT is for a single sequence of partial execution, which executes for the working set that fits into GPU memory as shown in Figure 3.

In order to generate LPT using PASs, VAST follows several steps. Considering that each PAS stores the list of pages accessed by a set of work-groups, the first step is to reduce several PASs, which will contain the list of pages accessed by multiple sets of work-groups. This reduction process is done until the accessed page size in the reduced PASs for all global arguments does not exceed the physical memory size. Algorithm 1 illustrates how VAST reduces the PASs. As illustrated in the algorithm, the outermost loop (line 5-20) iterates over the sets of work-groups, and the PAS used by a set of work-groups are OR-reduced to the PAS used by the next set of work-groups for each global argument (at line 9). As a result, OR-reduced PASs will contain the pages accessed by the multiple sets of work-groups. After OR-reduction, VAST checks the number of accessed pages by executing SUM-reduce on reduced PAS (line 10-11). If the total page size of the reduced PAS exceeds the GPU memory, it stores the range for partial execution (line 13-19), and continue reduction until PASs for all the work-groups are processed.

The time complexity of this algorithm is \( O(k \times N \times M) \), where \( k \) is the number of global arguments, \( N \) is the number of work-groups, and \( M \) is the number of PAS entry count. In general, \( k \) is a very small number (<5) and \( N \) can be reduced by letting several work-groups to share one PAS. Furthermore, reduction operation, which takes \( O(M) \), can be further optimized by utilizing SIMD instructions as reducing two entries is independent from reducing other entries. In other words, 16 entries can be reduced at once with the 128-bit SIMD instruction as each PAS entry consists of a single byte. As a result, the entire PAS reduction produce negligible overhead, which is evaluated in Section 5.

Once reduced PASs were computed, VAST keeps reduced PASs and the ranges of work-groups for the partial execution. Each reduced PAS that corresponds to the range will represent a list of pages that will be accessed by the range of work-groups to be executed.

With reduced PASs, the next step is to allocate the frame buffer for each global argument on the host. At this point, VAST knows how much memory space is required for the frame buffer by multiplying the page size by the number of valid \( \text{(TRUE)} \) entries in the reduced PAS. Finally, VAST starts to fill up the LPT and the frame buffers by iterating entries of the reduced PASs. If the entry value of PAS is \( \text{TRUE} \), it copies the corresponding page of the plain array to the contiguous memory space, the frame buffer. At the same time, the location (index) of the page frame in the frame buffer is stored to the entry of LPT as shown in Figure 6. One intelligent feature is that if the corresponding argument is a \textit{Write-Only} argument, VAST fills up the frame buffer only for the shared pages, which will be discussed in detail in the next section.

4.4 Forwarding Shared Pages

As discussed in Section 2, the OpenCL uses a relaxed memory consistency model for global memory within NDRange. Thus, any order of work-group execution will produce the same result without concerning write-conflicts among work-groups. However, in VAST, the granularity of memory transaction between the host and the GPU is a page frame, which is much larger than usual memory write. As a result, false sharing of a page may occur among the work-groups with regards to a writable frame buffer because more than one work-group can change the same page even though they write different location within the page. Sharing the same pages between work-groups within the same sequence of partial execution is safe because there is no write-conflict within a page as discussed. However, if a page was shared among work-groups in different sequences of partial execution, VAST must transfer up-to-date page frames for each partial execution due to Write-After-Write (WAW) dependencies. One option is that VAST waits for the partial out-
put to be arrived from the previous partial execution, and also waits for the partial output to be recovered to the plain array. After that, VAST proceeds with LPT and frame buffer generation in order to transfer up-to-date page frames for the next partial execution.

Obviously, this option brings a serious serialization of the execution, because LPT and frame buffer generation could be overlapped from the kernel execution and transferring data back and forth between the host and the GPU as shown in Figure 3. In order to avoid serialization, VAST precomputes shared pages among each sequence of partial execution, and selectively copies the shared pages by looking at the list of shared pages (Shared PAS). A shared PAS for partial execution can be precomputed using reduced PASs shown in Figure 7. The example in the figure illustrate that there are four reduced PASs, which means the entire kernel execution was decomposed into four sequences of partial execution. At each sequence, every accessed page in the PAS must be compared with the same page in previous sequences of PAS. For example, at the fourth sequence D, VAST performs OR-reduce PASs for all previous sequences, A to C, which represents all the pages accessed by the previous sequences of partial execution. After that, VAST AND-reduce it with the PAS of the fourth sequence. VAST also keeps these shared PASs only for the write arguments, and selectively copies the frame buffer when the partial results are transferred back to the host.

4.5 Transfer Optimization

While VAST performs several sequences of partial execution, the working set between consecutive sequences can be exactly same. In this case, VAST does not have to regenerate the LPT and frame buffer for the next sequence, since those are already present in the GPU memory from the previous sequence. In order to remove duplicated transfer between partial executions, VAST also compares the reduced PAS with the previous reduced PAS before generating LPT and frame buffer. The comparison is done using XOR-reduction between two PASs. If all the entries of XOR-reduced PAS are filled with TRUE, it means the pages required by the next partial execution is exactly the same as the previous one. With this optimization, VAST can further improve performance, and this optimization is evaluated in Section 5.4.

5. EVALUATION

Implementation: VAST was prototyped using Clang [1] for OpenCL front-end, and Low-Level Virtual Machine (LLVM) 3.3 [19] for the back-end. Once Clang parses the OpenCL kernel and generates the IRs, LLVM transforms it to the inspector kernel and the paged access kernel. After transformation is done, it lowers IRs to PTX binary to launch the kernels in the GPU. In order to evaluate an OpenCL kernel with indirect memory accesses, VAST also transforms the kernel into inspector code in x86 for PAS generation in the CPU.

Baseline: For the experiments, we used a real machine configured as shown in Table 1. We used the OpenCL execution on the CPU device as our baseline under the assumption that OpenCL kernels are being carried back to the CPU device (OpenCL's logical device) when the working set size exceeds the GPU's memory. This is possible because the CPU device, which shares the address space with the host, has 32 GB of memory, and thus it can execute on a large working set. For the CPU execution, we used the Intel OpenCL library [2], which fully utilizes all cores with simultaneous multi threading (SMT) and SIMD instructions (SSE and AVX). In addition, the Intel OpenCL library allows a kernel to access the host's memory directly if the buffer was created with the CL_MEM_USE_HOST flag. Therefore, our baseline does not include data transfer time.

Also, we measured wall clock time on VAST system that includes PAS, LPT, and frame buffer generation as well as buffer transfer, paged access kernel execution, and array recovery.

Benchmarks: For the evaluation, we used the benchmarks from NVIDIA Computing SDK [3]. We bailed out very simple benchmarks such as BandwidthTest, InlinePTX, VectorAdd, etc, but we added sparse vector matrix multiplication (SpMV) from our implementation in order to evaluate the kernel with indirect memory accesses. Because the benchmark suite is for evaluating NVIDIA GPUs, some benchmarks are unable to take large inputs. For these benchmarks, we only modified them to take large inputs. The list of applications, execution parameters, and working set size are shown in Table 2. To evaluate the scalability of the GPU, we used a working set size of more than GPU memory size (2 GB).

GPU with infinite memory (InfGPU): We also estimated the performance of a hypothetical GPU which is assumed to have infinite physical memory. In order to estimate kernel execution time on InfGPU, we measured the baseline (CPU) kernel execution time for a workload of 2 GB, which is the limit of the real GPU memory. Next, we compare this execution time with the baseline execution time for the actual size of workload as defined in Table 2. By comparing these two execution times, we computed scaling factor of kernel execution time. After that, we ran 2 GB of workload on the real GPU, and multiplied the kernel execution time by the scaling factor. Meanwhile, in order to estimate the transfer time for actual workload, we also measured the time for 2 GB of data transfer to the real GPU, and then extrapolated it to the actual size as shown in Table 2.

5.1 Results

First, we evaluated the speedup of VAST system over Intel OpenCL execution varying page size as shown in Figure 8. As shown in the figure, VAST performs better on every benchmark, geometric mean of 2.6x speedup with 4KB page frame, even though it involves the time for LPT generation, buffer transfer time, and array recovery. In general, as shown in the figure, reducing the page frame size brings slowdown on both PAS and LPT generation due to increased number of PAS entries. However, having too large a page frame size (8KB) may also have a negative effect as it may increase internal fragmentation, the ratio of unused data in a page frame. More internal fragmentation produces unnecessary data copy between plain array to the frame buffer, and thus additional buffer transfer between the host and the GPU.

In Figure 8, most benchmarks showed a performance gap between VAST and InfGPU. The gap generally comes from the exposed cost of generating the first LPT and frame buffer as well as the last array recovery (prolog and epilog of pipelined execution), which is not necessary for InfGPU. On the contrary, for compute-intensive benchmarks, such as MatrixMul and NBody, VAST performs as good as InfGPU. The main reason is that these compute-intensive kernels have smaller working sets, which makes the exposed cost negligible.
<table>
<thead>
<tr>
<th>Application</th>
<th>Execution Parameters</th>
<th>OpenCL Kernel</th>
<th>Buffer Size</th>
<th># of Work-groups</th>
<th># of Work-groups / PAS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Input</td>
<td>Output</td>
<td>Total</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>256 million options</td>
<td>BlackScholes</td>
<td>3.0 GB</td>
<td>2.0 GB</td>
<td>5.0 GB</td>
</tr>
<tr>
<td>FDTD3d</td>
<td>3D dimsize=1024, Radius=2</td>
<td>FiniteDifferences</td>
<td>4.0 GB</td>
<td>4.0 GB</td>
<td>8.0 GB</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>20,480×20,480 matrices</td>
<td>MatrixMul</td>
<td>3.1 GB</td>
<td>1.6 GB</td>
<td>4.7 GB</td>
</tr>
<tr>
<td>MedianFilter</td>
<td>30,720×17,820 PPM image</td>
<td>ckMedian</td>
<td>2.0 GB</td>
<td>2.0 GB</td>
<td>4.0 GB</td>
</tr>
<tr>
<td>MersenneTwister</td>
<td>1.15 billion numbers</td>
<td>MersenneTwister</td>
<td>1 MB</td>
<td>4.3 GB</td>
<td>4.3 GB</td>
</tr>
<tr>
<td>BoxMuller</td>
<td>41 million particles</td>
<td>BoxMuller</td>
<td>4.3 GB</td>
<td>4.3 GB</td>
<td>8.6 GB</td>
</tr>
<tr>
<td>Reduction</td>
<td>940 million options</td>
<td>Reduc6</td>
<td>1.1 GB</td>
<td>1.3 GB</td>
<td>2.5 GB</td>
</tr>
<tr>
<td>SobelFilter</td>
<td>30,720×17,820 PPM image</td>
<td>ckSobel</td>
<td>3.5 GB</td>
<td>64 KB</td>
<td>3.5 GB</td>
</tr>
<tr>
<td>SpMV</td>
<td>8 M x 64 matrix, 4 M vector</td>
<td>SparseMV_ELL</td>
<td>2.0 GB</td>
<td>2.0 GB</td>
<td>4.0 GB</td>
</tr>
</tbody>
</table>

Table 2: Benchmark Specification

Another point from Figure 8 is that speedup of SpMV increases as the size of the frame grows. The main reason is that PASs are generated using the CPU due to indirect memory accesses, while the performance of generating PASs in the CPU are heavily dependent on the size of PASs due to cache size of the CPU. In detail, the sparse matrix is originally an 8M×4M dense matrix and it is converted to an 8M×64 sparse matrix, with another 8M×64 matrix that contains non-zero column indices (index matrix). For 64 columns of the index matrix, we put random numbers among 8 millions, which is the actual width of the dense matrix. During PAS generation, PASs for input vector will be accessed very frequently among work-groups. Because non-zero column index values are randomly used within 8 million, having small PAS will have higher probability of fitting into lower level cache. To illustrate, for 4GB input matrices as shown in Table 2, 2KB page frame size results in 2 MB of PAS, while 8KB page frame size requires only 512KB of PAS.

5.2 Execution Time Breakdown

In order to analyze detailed behaviors of VAST for each benchmark, we measured the time for each stage of VAST system from PAS generation to array recovery as discussed in Section 3 and 4. Figure 9 illustrates the timeline of VAST execution for each benchmark using 4KB page frame size. In the figure, each color corresponds to a sequence of partial execution. To explain the result, PAS gen includes the time spent in the inspector kernel to generate PASs and also involves the time for transferring PASs to the host. PAS reduction shows the time to compute reduced PASs for partial execution. Through this step, PASs for all sequences of partial executions are ready. Next, PAS/Frame gen illustrates the time to generate LPT using PASs and copying the working set from the plain array to frame buffers. While the first LPT and frame buffer are generated, shared PASs for the next partial execution are computed in background. HostToDev is the time spent in transferring LPTs and frame buffers of the input arguments to the GPU, but also includes the time for forwarding shared page frames of the output arguments from the previous partial execution. Last, Kernel Exe. represents paged kernel execution time, and Recovery is for the time to copy page frames to the output array in the host.

As shown in the figure, PAS gen and PAS reduction, the initial cost, takes a small amounts of time out of the entire execution, except for SpMV. The reason why SpMV shows large overhead on PAS gen is that PASs are generated using the CPU due to indirect memory accesses. Even though PAS generation was parallelized using multiple threads on the CPU, the cost of PAS gen overwhelms other parts of execution. However, VAST still outperforms CPU execution on SpMV as shown in Figure 8.

While SpMV indirectly accesses the real arrays by simply loading the index from the index array, there could be a case where...
the computation of the index is complex. In this case, VAST still forces the CPU to generate PASs even if actual computations are simple. As a result, most of the execution time is spent in PAS generation on the CPU. Therefore, if this pattern is detected, it is more desirable to revert the entire computation back to the CPU.

On the other hand, the execution time of compute-intensive benchmarks, such as MedianFilter, MatrixMul, and Nbody, is heavily dependent on the kernel execution time. Especially for MatrixMul and Nbody, the initial cost of generating LPT and the cost of transferring data are almost invisible from the entire execution time. The reason why MatrixMul performs lots of partial executions is that a small number of work-groups require the maximum number of physical memory space (2 GB). The detailed behavior of MatrixMul is discussed in Section 5.4.

Other than SpMV and compute-intensive benchmarks, every benchmark shows that there is no critical bottleneck during partial executions, as transferring partial working set from the host to the GPU (HostToDev) starts right after getting partial results back from the GPU (DevToHost). In some benchmarks, however, HostToDev waits for LPT/Frame gen in the later sequences of partial execution. The reason why LPT/Frame gen and Recovery takes considerable amount of time is mainly due to copying frame buffer within the host memory, which has less bandwidth compared to others. This latency will decrease in the future as DRAM bandwidth increases (e.g. DDR 4). Meanwhile, this delay can be also hidden if VAST uses triple-buffering for the LPT and the frame buffers, which would require more preserved memory space in the host.

In Figure 9(e) for MersenneTwister, one thing to notice is that LPT/Frame gen takes some time only for the second and third par-
Figure 10: Paged access kernel execution time normalized to the original kernel execution time. Working set size for each benchmark is approximately 1.6GB which does not exceed the physical memory size. This illustrates page lookup overhead during kernel executions.

5.3 Page Lookup Overhead

As discussed in Section 3 and 4, if the working set size exceeds GPU’s memory size, VAST transforms the OpenCL kernels to make them access the data through LPT. As a result, one additional memory access occurs for each global memory access for page lookups. In the worst case, VAST could experience 2x slowdown due to doubled memory accesses for looking up pages. However, a realistic lookup overhead will not be significant because the size of LPT is small enough to fit in the GPU’s cache and all the work-items in a work-group have a higher probability of looking up the same page, taking advantage of memory level parallelism. In order to evaluate accurate page lookup overhead, we forced VAST to execute paged access kernels on a small working set, and compared the time spent in the GPU for the kernel execution. For the overhead estimation, we chose our working set size as approximately 1.6GB to not exceed the physical memory size on the GPU.

Figure 10 illustrates paged access kernel execution time, normalized to normal kernel execution time. As shown in the figure, page lookups bring 13.4% overhead on average with 4KB page size. A majority of the benchmarks shows less than 10% page lookup overhead, but FDTD3d shows considerable overhead. FDTD3d is a 3-dimensional stencil benchmark in which 3-dimensional adjacent input elements must be accessed to compute one output element. In this case, working sets of adjacent work-items are not contiguous in linear array space, thus VAST cannot benefit from reorganized page frames, causing noticeable overhead. In the meantime, VAST shows better performance on NBody by taking advantage of the memory level parallelism from gathered pages as discussed in Section 5.1.

Similarly, VAST performs better on BoxMuller with an 8KB page size as shown in Figure 10. The reason why BoxMuller shows a considerable performance difference upon varied page sizes is due to the memory access patterns of each work-item. BoxMuller was implemented to have each work-item access two memory locations with the distance of 128KB. With 4KB page size, two page lookups are necessary because the distance between two pages in the LPT is 32 elements (128 bytes). On the other hand, with an 8KB page size, the distance is reduced to 16 elements, thus two page frame indices can be brought into 128 bytes of a cache line at once. In addition to this cache effect, speedup on BoxMuller also comes from rearranging data in the contiguous page frames.

5.4 Case Study

In this section, we further investigate the performance of Matrix Multiplication (MatrixMul) on VAST since it is often used to measure the practical performance in GFlops for the system. We also discuss the case where the kernel has complicated indirect memory accesses.

In the implementation of MatrixMul from NVIDIA SDK, each work-item is in charge of generating one element of output matrix (matrix C). To generate one element, each work-item must access the entire row of source matrix A and the column of source matrix B. As a result, work-groups that are in charge of generating a set of output rows will require a set of rows of matrix A and the entire matrix B as shown in Figure 11(a). Based on this property, an expert programmer can manually modify the application to handle large matrix sizes that exceed GPU memory by following several steps. First, the program starts by getting the physical memory size of the GPU, and then checks if matrix B fits into the GPU memory. If it fits into the memory, the programmer makes it transfer matrix B to the GPU only once. After subtracting the size of matrix B from available memory size, the programmer calculates working set size for one work-group in order to get the number of rows per iteration based on the size of rest GPU memory. Once the number of rows for each iteration is calculated, the program executes the matrix multiplication kernel within a loop, and offset the matrix A and C to copy sub matrices (rectangular matrices) to the GPU device.
VAST automatically follows the exact same procedures above since it avoids duplicated data transfer as discussed in Section 4.5. Figure 11(b) illustrates the subset of the time frame in Figure 9(d). As shown in the figure, only the first transfer time (HostToDev) takes approximately 0.15 seconds, while next transfer time takes less than 0.03 seconds. This is because the entire matrix B is transferred to the GPU only once at the first transfer, while working set for matrix A and C are decomposed into partial rows and the partial rows are transferred to the GPU during rest of executions. Including every cost of VAST system, VAST achieves 155 GFlops for 20,480×20,480 matrices, while hand-modified code reached 171 GFlops. Nonetheless, the most significant advantage of VAST is that it can provide support for arbitrary type of OpenCL kernel, even with indirect memory accesses as described in the previous section.

6. RELATED WORK

A variety of research has been done to virtualize OpenCL devices as OpenCL programming models expose hardware details, such as the number of processing elements, the number of registers, and the physical memory size in each level of the hierarchy.

While a series of previous works tried to virtualize GPU hardwares in order to alleviate the efforts in optimizing performance [12, 29, 25], other types of researches focus on virtualizing multiple GPUs to distribute the workload automatically [21, 16, 17, 20, 18]. Although they split data parallel workloads into several parts to distribute them over multiple computing devices, they either limit the data parallel kernels to have specific memory access patterns [16, 17, 21, 18], or distribute the entire data to each device assuming the entire working set fits into each GPU memory [20]. On the contrary, VAST fully virtualizes the GPU memory that also supports kernels with indirect memory access pattern.

In the mean time, numerous works have proposed automatic CPU-GPU communication in order to remove explicit memory management [14, 13]. CGCM [14] uses compile-time type-inference to statically determine the types of data structures and transfer them between CPU and GPU memories, DyManD [13] employs runtime libraries to allocate data-structures in CPU and GPU memories. However, they only manage communications between kernels in cyclic patterns, assuming the entire working set fits into GPU memory. Larger data sets require explicit programmer management. More recently, OpenACC [6] and OpenHMPP [8] proposed new programming standards that consist of a set of compiler directives at a higher level, which hides hardware details from programmers. Despite these efforts, programmers must explicitly manage buffer transfer due to physical memory size on the GPU.

Inspector-executor (IE) model, which is applied to VAST, has been also adopted in the field of distributed systems [9, 22, 27]. In their works, the inspector code is used for identifying precise loop-carried dependencies for irregular access pattern in order to distribute loop iterations over multiple nodes. Although some of their works perform data compaction to reduce communication cost between nodes, the target system extracts the compacted data assuming that the target node has a plenty of memory space supported by virtual address space. On the other hand, VAST makes use of the inspector to generate LPT in order to provide virtual memory space on GPUs, while the executor is transformed from the original code to access through LPT.

Recently, there has been research projects to virtualize GPU memory [15, 23]. RSVM [15] is a region based virtual memory running on both CPU and GPU. In this work, programmers must define regions as the basic data unit abstractions. Then, RSVM manages these regions and transfers them if necessary. However, VAST does not need the programmer to think about access patterns of threads and divide the memory manually, thus, VAST is fully automatic and transparent. Pichai et al. [23] explored address translations on GPUs by putting translation look-aside buffers (TLBs) in the shader cores. Their work allows a unified virtual/physical memory space among CPUs and GPUs. However, they only target integrated CPU/GPU systems that share physical memory while VAST handles systems with discrete GPUs.

7. CONCLUSION

In this paper, we presented the VAST system that provides the programmer with an illusion of large memory space for OpenCL devices. Virtualizing memory space for the GPU is done by automatically partitioning the OpenCL’s NDRANGE into subsets of work-groups, and efficiently splitting the working set into several chunks required by a subset of work-groups based on available physical memory on the GPU. With the subsets of work-groups and divided working set, VAST performs partial execution multiple times consecutively.

To support these procedures, we introduced a new type of page table, LPT, which has addresses of page frames that will be accessed by a subset of work-groups on the GPU, and we introduced code transformation techniques to generate LPT efficiently and to make OpenCL kernels access memories through LPT.

Our experiments showed that NVIDIA GTX 760 GPU with 2 GB of memory successfully executed for more than 2 GB of working set regardless of memory access pattern. In addition, VAST achieved 2.6x speedup over CPU execution, which is a realistic alternative for large data computation.

8. ACKNOWLEDGEMENT

We thank the anonymous referees who provided excellent feedback. This research was supported by National Science Foundation grants CNS-0964478 and SHF-1217917, and the Defense Advanced Research Projects Agency (DARPA) under the Power Efficiency Revolution for Embedded Computing Technologies (PERFECT) program.

9. REFERENCES


