Energy-Efficient GPU Graph Processing with On-Demand Page Migration

Jacob M Hope
Texas State University
San Marcos, TX
jacobmhope@txstate.edu

Trisha Nag
Texas Tech University
Lubbock, TX
trisha.nag@ttu.edu

Apan Qasem
Texas State University
San Marcos, TX
apan@txstate.edu

***Abstract*—This paper presents a new approach to extracting improved performance-per-watt on large-scale hybrid graph applications with sparse data access patterns. The proposed technique takes advantage of demand paging, a technology recently introduced on CPU-GPU systems with heterogeneous memory. The strategy combines an analytical cost model, compiler transformations and a runtime system. The cost model, guided by runtime feedback, judiciously selects data structures for host placement which are migrated to the GPU during kernel execution via demand paging. We then introduce, two new code transformations, *kernel blocking* and *compute co-location*, to exploit page-level locality in host-resident data.**

We evaluate our strategy on four important algorithms in graph analytics: BFS, MST, SSSP and PageRank. Demand paging combined with kernel blocking causes significant reduction in PCIe traffic and yields an average speedup of 2.46, and up to a 5× performance improvement on BFS, over state-of-the-art methods. The performance boost does not incur a commensurate increase in GPU power draw, thereby leading to significant gains in energy efficiency. On average, 2.36 improvement in performance-per-watt is achieved across the four algorithms.

1. Introduction

HPC vendors have recently introduced technology that presents a unified view of multiple physical pools of memory contained within a compute node [1], [2], [3]. Unified Memory (UM) systems offer an obvious programmability benefit. Pointers can be freely used between different memory regions, relieving developers from the burden of explicitly managing data between *host* and *device*. Programmability benefits are particularly compelling for collaborative application design patterns such as those that have emerged for large-scale graph processing [4], [5], [6].

The performance benefits of UM are less obvious, however. Accessing host-resident data via demand paging can be expensive. Data is fetched over a high-latency, low-bandwidth channel (i.e., PCIe®) and page migrations incur additional overhead due to fault handling. Notwithstanding, the massively multi-threaded GPU kernels can potentially hide a large fraction of these latencies and mitigate the costs associated with host placement. Thus, a key challenge for hybrid memory systems is to determine the best placements for data structures such that data movement is minimized [7], [8].

In this paper, we present a strategy that leverages the on-demand page migration mechanism in UM systems to deliver improved energy efficiency on large-scale graph analytics applications. The performance boost does not incur a commensurate increase in power draw, thereby leading to significant gains in energy efficiency. Our solution exploits the following behavior that manifest in graph applications

(i) **GPU over-subscription:** Most real-world graph analytics workloads are too large to fit into the high bandwidth memory of a GPU [9]. This over-subscription problem is currently handled by partitioning the graph, an expensive step in and of itself, and invoking the GPU kernel multiple times on different partitions with some form of synchronization between invocations (e.g., C2GI and Pipelined patterns [6]).

(ii) **Segmented access:** Most algorithms will touch only a fraction of the graph in the average case. For example, a typical BFS instance will visit only half the nodes in the graph. This can be problematic on GPUs without a paging mechanism (pre NVIDIA Pascal). On such systems, the entire graph (or a partition) needs to be copied to the GPU before the kernel can be launched. This leads to gross under-utilization of the already constrained device memory.

(iii) **Sparse data access:** Graph analytics exhibit sparse data access patterns with low spatial locality [10]. This results in poor memory behavior on the GPU side. As nearby data is not accessed in close proximity in time, a page may be replaced before all of the data in the page has been fetched by the requesting SMs.

We observe that (i) and (ii) can be addressed by allocating data structures in host memory. Host-resident data is no longer constrained by device memory capacity, eliminating the GPU over-subscription problem (it is still constrained by CPU memory which is significantly larger). Moreover, since host-resident data is migrated on-demand when a GPU

This work was supported by the National Science Foundation through awards CNS-1253292, CCF-1659807 and OAC-1829644 and by equipment grants by IBM and Nvidia corporations

978-1-7281-5416-9/19 $31.00 © 2019 IEEE

request is received, for a given instance of an algorithm, only the segment of the data that is actually needed will be copied into device memory. Access sparsity cannot be dealt with placement alone. To tackle this issue, data locality issues must be considered. In particular, data locality must be analyzed within the migrated pages.

Our proposed solution builds on the above observations. First, we develop an analytical model, with runtime parameters as input, for intelligently selecting data structures for host placement. Second, we propose two new compiler optimizations, (i) *kernel blocking* and (ii) *compute co-location*. Kernel blocking exploits page-level locality of host-resident data. The key idea is to localize access to migrated pages such that (i) a particular warp accesses data from a designated page (or set of pages) and (ii) many requests to distinct pages are generated concurrently. Compute co-location offloads CPU tasks to the GPU to exploit page-level temporal locality. Finally, we develop auxiliary code transformations to reduce overhead associated with copy calls and kernel launch overhead in hybrid iterative applications.

2. Background and Motivation

2.1. Unified Memory

A heterogeneous memory system is typically partitioned into traditional DDR memory for the CPUs and high-bandwidth GDDR memory for the accelerators (e.g., HBM, MCDRAM). Unified Memory (UM) allows data in the CPU memory to be accessed directly from an accelerator without the need for an explicit copy. In a Unified Memory system [1], [2], [3], a GPU page fault occurs when a request is issued to data that is not resident in device memory. On a page fault, the UM driver allocates new pages on the GPU, requests the corresponding pages from CPU memory and the pages are migrated from host memory to device memory. The driver uses a prefetching heuristic to determine the number of pages migrated on a given request. On CUDA 10, the migration size varies between system page size and 2MB. During page fault handling, the GPU TLBs are locked to ensure each SM's view of memory is consistent. In a locked state, all new requests to the TLBs are stalled but outstanding requests can still move forward. Typically, page faults are serviced in *groups*. When the GPU generates multiple page faults concurrently (some of which may be to the same page), the UM driver removes duplicates, coalesces the requests and then transfers the data for all requests simultaneously. Thus, the performance penalty with respect to page fault handling, is not determined by the number of page faults but rather by the number of page fault groups. When data is migrated from host to device memory they are written to each of the higher level caches, creating potential for cache under-utilization. Once data is migrated, the CPU pages are freed and CPU page faults occur for subsequent requests to migrated data.

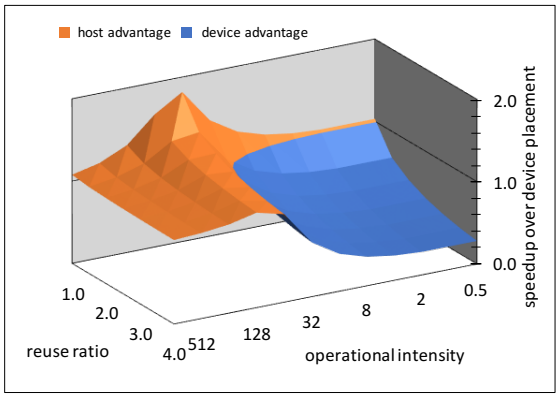

Figure 1: Performance trade-offs with demand paging. (system: Nvidia P100; Intel Xeon 12-core; PCIe® Gen 3)

2.2. Performance Trade-offs in Demand Paging

Latency hiding opportunities and data locality are two key considerations for demand paging in UM systems. We illustrate their performance impact with an example. For these experiments, we wrote a synthetic benchmark in the C2GI pattern [6]. In each invocation, the kernel performs a 2D stencil operation involving three arrays. The *operational intensity* of the kernel, an indicator of the latency hiding opportunities, is controlled via a compile-time parameter such that it can be invoked with intensity levels ranging from 1 to 1024. The amount of data reuse in device memory (*reuse ratio*) is also controlled via a an external parameter. We enforce this control by adjusting the size of the three arrays and the amount of data reuse exploited in device memory. Fig. 1 shows the performance space of the kernel as a function of operational intensity and reuse ratio. Performance is reported as the speedup obtained with demand paging over device placement. These results show that the performance space is multi-dimensional and non-linear. There is close to 10× performance difference between the best and worst points. In general, higher operational intensity and lower data reuse favors demand paging. But cusp effects are observed at different thresholds of operational intensity (e.g., 112 in Fig. 1) and reuse ratio.

3. A Cost Model for Host-resident Data

We construct an analytical model for making placement decisions at the individual data structure level. The model distills the fundamental performance trade-offs of host vs device placement in the following three parameters

(i) Host Access Penalty (ϕ) The additional time spent by a kernel in accessing a data structure d from host memory. ϕ is essentially the exposed latency due to accesses to d. It is the difference between the stalls introduced by the longer latency remote fetches and the fraction of those stalls hidden by the available computation in the kernel.

(ii) copy cost (τ) The copy cost represents the total cost incurred by an application due to explicit copies of

data between host and device memory. τ is a function of the copy time, the copy-to-computation ratio and the reuse ratio. Depending on the data access and application design pattern, the same data structure can be copied multiple times between the host and device, in either direction. τ refers to the total cost for these copies.

(iii) Overhead (σ) This component includes indirect overhead that is not captured in ϕ or τ. These overheads are implementation dependent and are independent of data access patterns. For example, the overhead associated with launching a GPU kernel, which varies across application paradigms.

A host-allocated data structure d, which is migrated via demand paging, will save on copy time (τ) and potentially reduce the overhead of multiple kernel invocations (σ). On the other hand, it will incur a penalty (ϕ) due to the longer latency access over a limited-bandwidth channel. Thus, we can estimate the performance gains for d as

$$P_d = \phi_d - \tau_d - \sigma_d \quad (1)$$

where P_d is net performance gains with host placement and σ_d is the difference in overhead between host and device placement.

The three parameters in (1) can be influenced by a multitude of inter-related factors. We identify a core set of application and system attributes that impact (1) and develop methods to estimate each.

```
in1 = malloc(size(in1));          // host
in2 = malloc(size(in2));          // host
out = deviceMalloc(size(out));    // device

copy(in1, size(in1), host_to_device);
copy(in2, size(in2), host_to_device);
kernel<<<grid,block>>>(in1, in2, out);
copy(out, size(out), device_to_host);
```

(a) before

```
// iterate over each data structure
// determine placement from model
for d in in1,in2,out
  isHost[d] = isHostResident(d,size(d), ...));

in1 = smartMalloc(in1,isHost[in1]);
in2 = smartMalloc(in2,isHost[in2]);
out = smartMalloc(out,isHost[out]);

if (!isHost[in1])
  copy(in1, size(in1), host_to_device);
if (!isHost[in2])
  copy(in2, size(in2), host_to_device);
new_grid = getGridSize(grid,in1,in2,out,isHost[]);

kernel<<<new_grid,block>>>(in1,in2,out);
if (!isHost[out])
  copy(out, size(out), device_to_host);
```

(b) after

Figure 2: Custom allocation and copy call elimination

4. Optimizations

In this section, we first describe our custom memory allocator, which places selected data structures in host memory for on-demand paging, and associated transformations. We then present the optimizations we developed to exploit opportunities created by host-resident data.

4.1. Custom Allocation

In the initial pass, the code is instrumented with calls to the runtime system that determines the suitability of on-demand migration of a data structure. For each data structure accessed by a GPU kernel, a single call to the `isHostResident()` function is inserted. For each such data structure, allocation and deallocation sites are identified and the calls are replaced with calls to a custom allocator called `smartMalloc()`. The custom allocator internally calls the appropriate CUDA functions (i.e., `cudaMalloc()` for device and `cudaMallocManaged()` for host) based on the result obtained from the `isHostResident()`.

4.2. Copy Call Elimination

Since host-resident data structures do not need to be explicitly copied to device memory, existing copy calls can be eliminated. The initial pass identifies the copy sites for each GPU-accessed data structure and wraps them in a condition clause such that copy calls are only invoked when the data structure is allocated to device memory. Fig. 2 outlines the custom allocation and the elimination of copy calls.

4.3. Reducing Kernel Launch Overhead

When data structures are placed in host memory, the need for multiple kernel invocations is reduced. In the extreme case, when all data is fetched via demand paging, the kernel needs to be invoked only once. Post host allocation, we apply a code transformation that determines the number of kernel invocations based on the distribution of data between host and device memory. The transformation employs a simple heuristic in which it aims to minimize the number of kernel invocations. Let, $\{d_0, ..., d_n\}$ be the set of data structures that are device mapped. Then the volume of data copied to device is $\sum_0^n size(d_i)$ and the number of times the kernel needs to be invoked is obtained as follows

$$k = \lfloor \sum_0^n size(d_i)/dev\_mem\_cap \rfloor + 1$$

The above formula guarantees that the kernel is invoked at least once even if no data is copied to device memory. The bounds of the loop that controls the kernel invocations is adjusted based on the value of k determined as above. The grid size and kernel parameters are also adjusted to match the number of calls. Fig. 3 shows before and after snapshot of this transformation.

```
in = malloc(size(in))
out = device_malloc(size(out));

// calculate # of iterations that
// 'fit' in device mem
n = (size(in) + size(out)) / DEV_CAP

// calculate buffer size
buf_size_in = size(in) / n;
buf_size_out = size(out) / n;

// invoke kernel in C2GI mode
for (i = 0; i < n; i++)
  copy_to_device(in + (i * buf_size_in), in_buf)
  kernel<<<grid,block>>>(in_buf, out_buf)
  copy_to_host(out_buf, out + (i * buf_size_out))
```

(a) before

```

in = smartMalloc(in, isHost[in]);
out = smartMalloc(out, isHost[out]);

dev_size = getDeviceAllocSize(in1,in2,out, ...);
iters = getIters(dev_size,DEVMEM_CAPACITY);
new_grid = getGridSize(grid,iters,in,out,isHost[]);
for (i = 0; i < iters; i++)
   if (!isHost[in])
     copy(in1, size(in1), host_to_device);
   kernel<<<new_grid,block>>>(in,out);
  if (!isHost[out])
     copy(out, size(out), device_to_host);
```

(b) after

Figure 3: Transformation to reduce kernel launch overhead

4.4. Kernel Blocking

Host-resident data affords new opportunities for improving memory access behavior. Notwithstanding, the data locality issues are significantly different than those encountered for device-mapped data structures. We discuss two of these issues here.

(i) Page-level locality: Since data is transferred from the CPU at page-level granularity, access to migrated data should be coordinated such that the kernel touches all needed data within the page, *before* the page is unmapped from the GPU.

(ii) Concurrent request to distinct pages: On the GPU, access to data must be coordinated such that nearby memory locations are accessed at the same time (i.e., by consecutive threads within a warp). This ensures that memory requests can be coalesced by the coalescing unit on the GPU. For host-resident data however, it is more profitable to request multiple distinct pages at the same time. This has two benefits. First, it allows the driver to overlap multiple fetches from system memory and hide their latencies. Second, at the time of servicing a page fault, the driver has more information about the access patterns, which leads to more effective page migration via prefetching.

We introduce a new code transformation called kernel blocking that addresses both (i) and (ii). The key idea is

```
in = smartMalloc(in, isHost[in]);
out = smartMalloc(out, isHost[out]);

kernel<<<grid,block>>>(in, out)

// kernel definition
kernel (in, out) {
   // body
            = in[tid];
   out[tid] =
}
```

(a) before

```
in = smartMalloc(in, isHost[in]);
out = smartMalloc(out, isHost[out]);

// total data set size obtained at runtime
dsize = get_size();

// blocking factor obtained at runtime
// from analytical model or via autotuning
blk_factor = get_blk(dsize, pagesize, grid, block);

// blocking parameter passed to kernel
kernel<<<grid,block>>>(in, out, dsize, blk_factor)

// kernel definition
kernel (in, out, blk) {
   // divide work among warps based on block size
   warps = get_warps(dsize, blk);
   // warp id is a function of block size
   // and grid dimensions
   wid = get_warp_id(blk, gridDim, blockDim);

   // loop injected in kernel body;
   // each warp performs multiple sweeps
   // within same region(page)
   #pragma unroll
   for (m = 0; m < blk; m++) {
     // data index obtained from block size
     // grid dimensions and other parameters
     index = get_index(wid, blk, m, warp_size);
        ... in[index];
     out[index] = ...;
  }
}
```

(b) after

Figure 4: Kernel blocking

to re-order the data access of host-resident data structures such that many distinct pages are requested by the warps within the same thread block. In addition, we apply *warp coarsening* [11], in which warps are *reused* to access the same page and exploit spatial locality withing a page. Fig 4 illustrates the kernel blocking transformation. The host-resident data structure is broken down into page size units. Each warp is then assigned to work on B pages where B represents the block size. With warp coarsening, each warp processes 32 (i.e., warp size) elements and then moves on to the next 32 within the same page. This behavior is affected by introducing a repeat loop inside the kernel as shown in Fig. 4(b), lines 25-29. The introduction of the repeat loop introduces implicit serialization while increasing warp granularity results in fewer active warps during kernel execution. Both of these can have an adverse effect on concurrency. We

```
0
1  initialize(in) {
2    for (i = 0; i < size(in); i++)
3      in[i] = ...
4  }
5
6  int main() {
7    in = smartMalloc(in,isHost[in]);
8    out = smartMalloc(out,isHost[out]);
9
10   initialize(in);
11
12   kernel<<<grid,block>>>(in, out)
13 }
```

(a) before

```
0
1  __global__ void initialize(in) {
2     tid = threadIdx.x + blockDim.x * blockId.x;
3      in[tid] = ...
4  }
5
6  int main() {
7    in = smartMalloc(in,isHost[in]);
8    out = smartMalloc(out,isHost[out]);
9
10   new_grid = getGrid(in);
11   initialize<<<grid,MAXBLOCK>>>(in);
12   kernel<<<grid,block>>>(in, out)
13 }
```

(b) after

Figure 5: Compute co-location

combat this problem by translating warp-level concurrency into ILP. The repeat loop inside the kernel is fully unrolled (Fig. 4(b), line 24) such that all iterations of the loop can progress concurrently. With the above transformation, access to a particular page is localized within a warp. This means that at any given point, concurrent warps will make requests to distinct pages, allowing page faults to be overlapped, thereby improving transfer times.

4.5. Compute Co-location

On a CPU-GPU heterogeneous compute node, access patterns to a shared data structure, d can be classified as (i) **host-first:** d is touched by a CPU thread and then by device kernel (typically input data, passed onto the kernel) and (ii) **device-first:** d is touched by device kernel and then by CPU thread (typically, output data). The memory management system on the latest NVIDIA GPUs implements a first-touch allocation policy [12]. In this scheme, memory is not physically allocated at the time the allocator is called but rather when the data is first accessed. If the GPU is first to touch a data element, then the page holding that element is allocated in device memory (and *vice versa* if the CPU accesses it first). This means that host-allocated, host-first data structure will incur page faults twice. Once on the CPU side at the time of its first access and then again when the data is accessed from the GPU. Both sets of page faults will negatively impact the overall application.

We developed a optimization called *compute co-location* to mitigate this problem. This optimization first identifies first-touch functions for all *host-resident*, *host-first* data structures. First-touch functions are those functions in which the data structure in question is first accessed on the CPU side. The first-touch function is then converted into a GPU kernel and the call to the function is replaced with a kernel launch statement. The grid size for this new kernel is determined from the size of the data structure. Fig. 5 shows an example transformation. In our current implementation only simple initialization routines are automatically converted while the rest are done manually.

5. Evaluation

5.1. Experimental Setup

We conduct experiments on a heterogeneous compute node featuring a POWER8 system with 128 logical cores connected to two NUMA sockets. The node has four NVvidia Tesla P100 GPUs connected to the CPU via NVLink. The environment Experiments is set up with CUDA 10.0 driver and runtime. Power draw for each SM on the GPU is measured with `nvprof` with `--system-profiling` turned on and the average power from all probes is reported here. Kernel execution times and data copy times are also measured with `nvprof` . Due to the wide variability of GPU page fault related metrics, each application run is repeated 100 times and only average values are reported after weeding out the outliers.

We evaluate our strategy on four key graph applications: (i) *BFS:* Breadth-first Search (ii) *SSSP:* Single-source Shortest Path (iii) *MST:* Minimum Spanning Tree and (iv) *PR:* page rank algorithm. *BFS*, *SSSP* and *MST* are obtained from the Lonestar benchmark suite [13] and we select the most optimal version for the experiments in this paper (e.g., Merrill et al. for BFS [14]). *PR* is obtained from the HeteroMark benchmark suite [6]. The input graphs are collected from Koblenz repository [15].

5.2. Data Movement

We first look at how demand paging impacts data movement over the CPU-GPU channel (NVLink). Fig. 6 shows the volume of data transferred via demand-paging in *BFS* for 20 input graphs. We compare this with the amount of data copied over NVLink when the graph is allocated in device memory. We observe that demand paging leads to substantial reduction in data traffic over the channel. The average reduction across 20 graphs is 36%. Even in the worst case, demand-driven paging is able to reduce data movement by at least 8% (*dblp*). The amount of reduction varies across graphs. This variability is a function of the size and the node degree distribution. Graphs with a higher variability in node degree (e.g., *wikipedia*), exhibit more

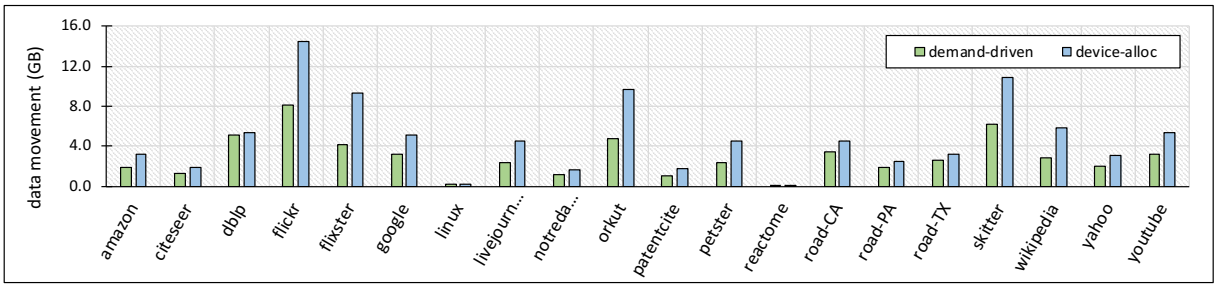

Figure 6: Reduction in data movement with demand-paging

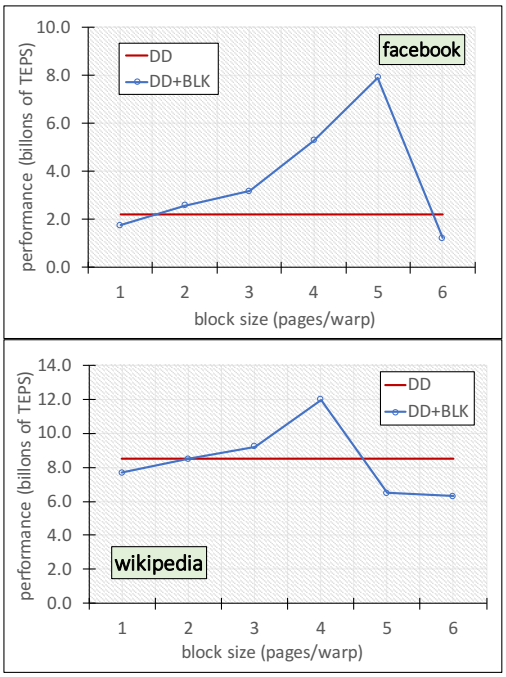

Figure 7: Impact of kernel blocking on performance

sparse data access within a thread block. With demand-paging, only the requested pages are migrated which leads to a more significant reduction in data movement.

5.3. Kernel Blocking

Fig. 7 shows the performance impact of the kernel blocking transformation. Performance numbers are shown for BFS across different block sizes for two input graphs with differing characteristics. As discussed in (§4.4), blocking factor refers to the number of pages assigned to each warp. To isolate the performance impact of kernel blocking, in Fig. 7, we report the performance of BFS without kernel blocking but with demand paging turned on. We observe that performance is quite sensitive to the choice of the blocking factor. For *facebook*, there is a factor of four performance difference between the best and the worst block sizes. In spite of this variation, a discernible performance pattern is detected. Very small block sizes (e.g., when warp access is localized to 1-2 pages) are generally not profitable. Similarly performance also degrades for larger block sizes. The sweet spot lies somewhere in between. We took an autotuning approach to discovering the optimal block size. In our experiments with 100 input graphs, we found the largest profitable block size for any graph to be 10 and the optimal block size fell within the range 2-5. These results suggest that although there is some performance variation, the space is not particularly large and profitable block sizes can be determined via tuning with relative ease.

5.4. Performance

Fig. 8 shows overall performance improvements achieved when kernel blocking is combined with demand-driven paging. Performance is measured in Traversed Edges Per Second (TEPS). Number of traversed edges is counted at runtime and is divided by the total execution of the application (not just the kernel). Fig. 8 reports the average TEPS achieved over 100 distinct graphs for the four graph algorithms we studied. We compare the performance of our strategy with the fully optimized CUDA version built with the following flags `nvcc -O3 -arch=sm_60 --ptxas-options -O3`. We observe that our technique yields significant performance improvements for all four applications. The most significant improvements are seen for *BFS* where demand-driven paging with kernel blocking attains a 4.59 speedup over CUDA. Demand-driven paging was more helpful for applications with higher operational intensity as it created more opportunities for latency hiding. For example, *MST* has a lower operational intensity than *SSSP*, leading to lower performance gains. Performance improvement were also affected by the *reuse ratio*. For instance in *PR*, although the kernel is invoked multiple times, data is copied to device only once. Thus, performance gains are considerably less.

5.5. Energy Efficiency

Finally, we discuss the energy efficiency of the proposed technique. As discussed, even though our technique can improve application performance it does not come at a cost of increased power draw. Most of the performance benefits come as a result of overlapping computation with

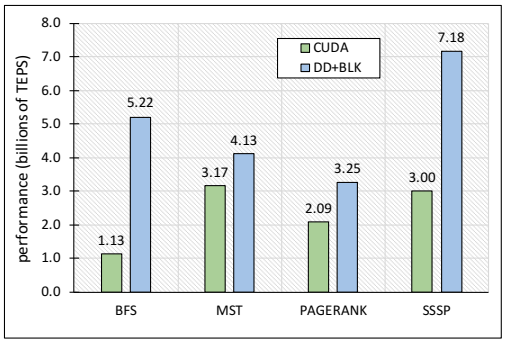

Figure 8: Performance improvements achieved with demand-driven paging (DD) and kernel blocking (BLK). Performance reported in billions of Traversed Edges Per Second (TEPS)

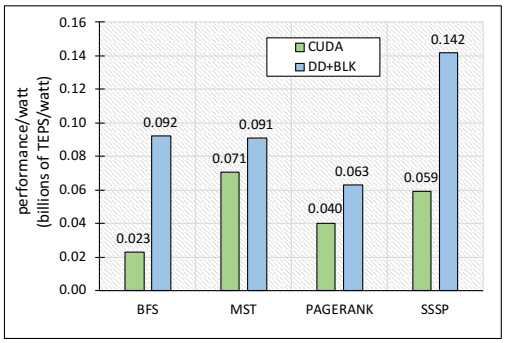

Figure 9: Energy efficiency with demand-driven paging (DD) and kernel blocking (BLK). Performance/watt is billions of Traversed Edges Per Second (TEPS) per watt

long latency host memory operations. This does not create undue pressure with respect to the activity levels on the SMs and hence keeps the power draw in check. Fig. 9 shows the energy efficiency of the four graph algorithms. We use performance-per-watt measured in TEPS/watt as a measure of energy efficiency. As we can see, the performance/watt numbers closely follow the performance numbers in Fig. 8. For *BFS* and *MST* there was a slight increase in average power draw (˜3 Watts). For *PR* and *SSSP* the power draw remained the same with and without demand paging.

6. Related Work

6.1. Heterogeneous Memory

There has been extensive research in mitigating NUMA effects on shared memory systems [16], [17], [18], [19]. This work addresses newer challenges in hybrid memory systems.

Introduction of new memory technology such as NVM and HBM has prompted a flurry of work in data organizations on heterogeneous memory systems [8], [20], [21], [22]. Yet, few have considered the performance issues that arise with data placement in CPU-GPU systems with a discrete device memory, which is the main focus of this work.

Shen *et al.* develop a placement strategy for the TI Keystone II DSP processor [22]. Their strategy shows substantial performance gains but is only applicable to SoCs and systems with physically shared memory. Chen *et al.* look at the problem of data placement *whithin* the GPU memory hierarchy (e.g., global, texture, or shared) and present coherence-free multiview, a framework that allows multiple views of a single data object to co-exist in GPU memory [23], [24]. Placement techniques leveraging the multiview exhibit impressive speedups. The multi-view approach is formalized PORPLE which addresses data placement problem [24]. Nonetheless, the framework is not extendable, in any direct way, to CPU-GPU Unified Memory systems.

Research in page migration on CPU-GPU systems have been limited to system-level approaches. Agarwal *et al.* develop an application-agnostic page migration scheme to maximize GPU throughput [25]. They extend this policy to incorporate runtime profiling and programmer guidance for allocation. Flores *et al.* looks at performance inefficiencies of demand paging and migration policies on systems with shared virtual memory [26]. Results show that the granularity of migration is an important consideration. Our proposed solution, which considers data placement at the application layer, can work in conjunction with system-level approaches to page migration.

6.2. Graph Analytics

There has been extensive work in optimizing graph analytics applications for the GPU. Many of these have focused on a single algorithm. Bisson et al. [27] employs sparse linear algebra to optimize the PageRank algorithm. They propose a method of overlapping kernel execution with data transfer between CPU and GPU. However, they do not consider they effect of dynamic page migration.

Merrill et al. [14] develop a high-performance BFS by using a prefix scan for the frontier computation and an improved scheduling heuristic for task management. Beamer et al. introduced the direction-optimization strategy for BFS [28]. Variants of this technique have been used in several other works in optimizing BFS on single [29] and multi-GPU systems [9]. Davidson et al. propose the Near-Far optimization for SSSP [30]. McLaughlin and Bader propose a hybrid betweenness centrality on GPUs, which can switch between the work-efficient mode and edge-parallel mode based on the change of vertex frontiers [31].

Broader performance issues in GPU graph processing have also been studied but not as extensively. Che et al. conduct a comprehensive study of the irregularity of GPU graph algorithms [32]. Wu et al. and Kaleem et al. have investigated the impact of GPU synchronization and load balancing [33]. Li et al. look at data locality in GPU kernels and propose a technique for improving locality by reducing synchronization overheads [34]. Although many sophisticated techniques for GPU graph processing have been developed, the impact of demand-driven page migration have not been investigated. The techniques described in this paper are complementary to existing techniques and

will in general lead to further performance improvements. For example, we see significant gains with a demand paging implementation of Merril-BFS [14] (§ 5.4).

7. Conclusions

This paper presented a new strategy for orchestrating data movement in over-subscribed, irregular graph applications. The technique leverages the demand paging mechanism available on UM-enabled CPU-GPU systems. We develop a cost model for selecting data structures for demand paging and propose a new compiler optimization for improving data locality of host-resident data. Experimental results on four key graph algorithms show that the proposed strategy can yield integer factor performance improvements across a range of input graphs with varying characteristics. Because performance gains come primarily from the reduction in data movement, the technique has minimal negative impact on the power draw which leads to a 2.36× improvement in performance-per-watt on average.

References

[1] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[2] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[3] AMD, "Radeon Open Compute Manifest 1.6," https://https://rocm.github.io/, accessed: 2019-04-09.

[4] J. Gómez-Luna, I. El Hajj, L.-W. Chang, V. García-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: collaborative heterogeneous applications for integrated-architectures," in *ISPASS*, 2017, pp. 43–54.

[5] K. Krommydas, W.-c. Feng, C. D. Antonopoulos, and N. Bellas, "Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures," *Journal of Signal Processing Systems*, vol. 85, no. 3, pp. 373–392, 2016.

[6] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. R. Kaeli, "Hetero-mark, a benchmark suite for CPU-GPU collaborative computing," in *IISWC*, 2016.

[7] A. Qasem, A. Aji, and G. Rodgers, "Characterizing data organization effects on heterogeneous memory architectures," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Mar 2017.

[8] S. Wen, L. Cherkasova, F. X. Lin, and X. Liu, "Profdp: A lightweight profiler to guide data placement in heterogeneous memory systems," in *ICS*, 2018, pp. 263–273.

[9] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-gpu graph analytics," in *IPDPS*, 2017, pp. 479–490.

[10] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on gpus," in *IPDPS*, 2013, pp. 463–474.

[11] A. Magni, C. Dubach, and M. F. P. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *SC*, 2013.

[12] NVIDIA, "Cuda c programming guide," 2017.

[13] "LonestarGPU," http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu.

[14] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *PPoPP*, 2012.

[15] "The Koblenz Network Collection: KONECT," http://konect.cc, accessed: 2019-07-15.

[16] I. S. Barrera, M. Moretó, E. Ayguadé, J. Labarta, M. Valero, and M. Casas, "Reducing data movement on large shared memory systems by exploiting computation dependencies," in *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 2018, pp. 207–217.

[17] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: a holistic approach to memory placement on numa systems," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 381–394, 2013.

[18] A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen, "Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, p. 30, 2014.

[19] M. Tanaka and O. Tatebe, "Workflow scheduling to minimize data movement using multi-constraint graph partitioning," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012, pp. 65–72.

[20] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 15.

[21] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "Rthms: A tool for data placement on hybrid memory system," in *ACM SIGPLAN Notices*, vol. 52, no. 9. ACM, 2017, pp. 82–91.

[22] D. Shen, X. Liu, and F. X. Lin, "Characterizing emerging heterogeneous memory," in *ISMM*. ACM, 2016, pp. 13–23.

[23] G. Chen and X. Shen, "Coherence-free multiview: Enabling reference-discerning data placement on GPU," in *ICS*, 2016.

[24] G. Chen, X. Shen, B. Wu, and D. Li, "Optimizing data placement on gpu memory: A portable approach," *IEEE Trans. Comput.*, vol. 66, no. 3, Mar. 2017.

[25] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for gpus within heterogeneous memory systems," in *ASPLOS*, 2015, pp. 607–618.

[26] V. Garca-Flores, E. Ayguade, and A. J. Pena, "Efficient data sharing on heterogeneous systems," in *2017 46th International Conference on Parallel Processing (ICPP)*, Aug 2017, pp. 121–130.

[27] M. Bisson, E. Phillips, and M. Fatica, "A cuda implementation of the pagerank pipeline benchmark," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–7.

[28] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *Supercomputing (SC)*. IEEE, 2012, pp. 1–10.

[29] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on gpus," in *Supercomputing (SC)*. IEEE, 2015, pp. 1–12.

[30] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *IPDPS*, 2014, pp. 349–359.

[31] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the gpu," in *SC*, 2014, pp. 572–583.

[32] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *IISWC*, 2013, pp. 185–195.

[33] R. Kaleem, A. Venkat, S. Pai, M. Hall, and K. Pingali, "Synchronization trade-offs in gpu implementations of graph algorithms," in *IPDPS*. IEEE, 2016, pp. 514–523.

[34] A. Li, W. Liu, L. Wang, K. Barker, and S. L. Song, "Warp-consolidation: A novel execution model for gpus," in *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 2018, pp. 53–64.