XPlacer: Automatic Analysis of Data Access Patterns on Heterogeneous CPU/GPU Systems

Peter Pirkelbauer*†, Pei-Hung Lin*, Tristan Vanderbruggen* and Chunhua Liao*
*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
Email: {pirkelbauer2, lin32, vanderbrugge1, liao6}@llnl.gov
†Department of Computer Science, University of Central Florida, Orlando, FL 32816, USA

***Abstract*—This paper presents XPlacer, a framework to automatically analyze problematic data access patterns in C++ and CUDA code. XPlacer records heap memory operations in both host and device code for later analysis. To this end, XPlacer instruments read and write operations, function calls, and kernel launches. Programmers mark points in the program execution where the recorded data is analyzed and anomalies diagnosed. XPlacer reports data access anti-patterns, including alternating CPU/GPU accesses to the same memory, memory with low access density, and unnecessary data transfers. The diagnostic also produces summative information about the recorded accesses, which aids users in identifying code that could degrade performance.**

The paper evaluates XPlacer using LULESH, a Lawrence Livermore proxy application, Rodina benchmarks, and an implementation of the Smith-Waterman algorithm. XPlacer diagnosed several performance issues in these codes. The elimination of a performance problem in LULESH resulted in a 3x speedup on a heterogeneous platform combining Intel CPUs and Nvidia GPUs.

***Index Terms*—GPGPU, heterogeneous systems, high-performance computing, code instrumentation**

I. Introduction

Modern high-performance computers often combine central processing units (CPUs) and general-purpose graphics processing units (GPGPUs and GPUs) to improve peak performance and meet energy-efficiency requirements. As of November 2019, 136 of the Top 500 supercomputers are accelerated with Nvidia GPUs [1]. The resulting heterogeneous compute nodes exhibit a complex memory architecture.

Nvidia GPUs are frequently programmed using CUDA [2], an extension to the ISO C++ programming language [3]. In the past, CUDA programmers had to manually transfer data between CPU and GPU. In addition, programmers also had to carefully select the kind of device memory where the data should be placed in. Options included global and local memory, texture memory, constant memory, shared memory, and registers. The right choice of memory depended on data size and data access characteristics. Newer GPU generations are less sensitive towards data placement choices [4]. Although explicit data transfer and data placement can be cumbersome to use, they remain popular for attaining good performance.

Recently, Nvidia started supporting a unified memory model [5], where the CUDA runtime system automatically manages data transfer and data placement depending on how applications access memory. Programmers can use hints to indicate how (*e.g.*, read mostly) and where (*e.g.*, host, device) data is likely to be used. While CUDA's unified memory model simplifies combined CPU and GPU programming, the correct and efficient use remains non-trivial. For example, providing a hint that is inconsistent with actual data use may degrade performance. The detection of correctness issues or performance bottlenecks requires deep insights into a program's execution. We call memory access patterns that are prone to correctness and performance bugs *anti-patterns.*

This paper introduces XPlacer, a source-code instrumentation framework and first runtime system to detect memory access anti-patterns in combined CPU/GPU code. XPlacer's instrumentation is developed as a plugin of the ROSE source-to-source transformation infrastructure [6], [7]. The instrumentation wraps memory operations (*e.g.*, reads, writes), function calls, and kernel launches in a custom API. XPlacer instruments both host and device code. XPlacer's runtime system implements the API and keeps track how CPU and GPU access memory with the goal to identify inefficient or erroneous codes. For example, a memory location that is frequently read by CPU and GPU may incur latencies through unnecessary data transfers. XPlacer's diagnostic helps skilled programmers uncover and fix such subtle issues.

The contributions of this paper are:

1) a description of common CPU/GPU memory access anti-patterns,
2) a design of an instrumentation API to capture the necessary information to recognize three representative anti-patterns of CPU/GPU memory accesses,
3) a design of runtime algorithms to automatically detect the selected anti-patterns,
4) an implementation of a framework consisting of both a code instrumentation tool and a runtime library to effectively identify anti-patterns,
5) the evaluation of our framework using a set of benchmarks and an HPC proxy application.

The remainder of the paper is organized as follows: §II provides the background of this work, §III describes the design principles and implementation of the instrumentation framework and the prototype runtime library in detail, §IV evaluates our approach on available benchmarks and proxy applications, §V gives an overview of related work, and §VI concludes and offers some ideas for future work.

II. Background

This section offers background on Nvidia's unified memory and LULESH, an HPC proxy application developed at the Lawrence Livermore National Laboratory [8], [9].

A. Nvidia GPUs and Unified Memory

The memory management among CPUs and GPUs has been a complicated problem and is critical to performance and energy efficiency. Before CUDA 6.0, the data shared by CPU and GPU had to be allocated into separate memories, which required explicit memory copy calls to complete data migration. Since CUDA 6.0, Nvidia has introduced unified Memory (UM). UM provides a single, coherent shared memory view within heterogeneous architectures using CPUs and GPUs. It allows all processors to see a single memory image with a common address space and avoids the overhead of explicit data migration [10]. With a single memory place for all CPUs and GPUs within a heterogeneous architecture consisting of separated physical memory spaces, UM relieves programmers from manual management of data migration between CPUs and GPUs such as inserting memory copy calls and deep copying pointers. It tremendously improves productivity and also enables oversubscribing GPU memory.

Nvidia continuously improves UM throughout different generations of GPUs. The latest UM implementation has accumulated a rich set of features including GPU page fault counts, on-demand migration, over-subscription of GPU memory, concurrent access and atomics, access counters, etc.

B. The cudaMemAdvise API

To enable better performance of UM, CUDA allows developers to give the UM driver additional advice about how to manage a given GPU memory range via `cudaMemAdvise(const void*, size_t, cudaMemoryAdvise, int)` [11]. The first two parameters of this function accept a *pointer* to a memory range with a specified size. The memory range should be allocated via `cudaMallocManaged` or declared via variables allocated in `__managed__` memory [11]. The third parameter sets the advice for the memory range. The last parameter indicates the associated device's id. The details and differences of these four kinds of advice are presented as follows:

- `Default`: This represents the default on-demand page migration to accessing processor on first touch.
- `cudaMemAdviseSetReadMostly`: This advice is used for the data which is mostly going to be read from and only occasionally written to. The UM driver may create read-only copies of the data in a processor's memory when that processor accesses it. If this region encounters any write requests, then only page where the write occurred will be valid and other copies will be invalidated.
- `cudaMemAdviseSetPreferredLocation`: This advice sets the preferred location to either a GPU device or to host memory. Setting the preferred location does not cause data to migrate to that location immediately. The policy only guides what will happen when a fault occurs on the specified memory region: if data is already in the preferred location, the faulting processor will try to directly establish a mapping to the region without causing page migration. Otherwise, the data will be migrated to the processor accessing it if the data is not in the preferred location or if a direct mapping cannot be established.
- `cudaMemAdviseSetAccessedBy`: This advice implies that the data will be accessed by a device. The device option includes both CPU and GPU. It has no impact on the data location and will not cause data migration. It only causes the data to be always mapped in the specified processor's page tables, when applicable. The mapping will be updated if the data is migrated. This advice is useful to indicate that avoiding faults is important for some data, especially when the data is accessed by a GPU within a system containing multiple GPUs with peer-to-peer access enabled.

The effect of `cudaMemAdvise` can be reverted with the following options: `UnsetReadMostly`, `UnsetPreferredLocation`, and `UnsetAccessedBy`.

C. Performance Impact: LULESH RAJA Example

Given the rich memory management APIs provided by CUDA, programmers are facing multiple choices to manage their data on GPU memory. These choices have significant impact on the final performance. We use a concrete example program to illustrate such impact.

LULESH is a hydrodynamics code to numerically solve Sedov blast problems. The latest LULESH version is based on RAJA [12]. RAJA is a portable template-based application development framework where computational kernels are expressed as lambda functions. RAJA can be configured to run the computational kernels using different parallel execution strategies (*i.e.*, OpenMP and CUDA).

At its core, LULESH uses a singleton class `Domain` that encapsulates all data needed by computational kernels on CPU and GPU. `Domain` contains pointers to dynamically allocated arrays. The arrays store data for the numeric computation. The domain object and most of its data arrays are allocated at program start using `cudaMallocManaged`. Most of the object structure is stable during the program execution. In the non MPI-version, the CPU and GPU access a disjoint data set.

In addition, the class `Domain` contains pointers to temporary data that are also allocated in unified memory. The temporary memory is allocated by the CPU but used by the GPU for computation. The memory is allocated and freed twice during each timestep.

On Intel x86 systems with a Pascal or Volta GPU, accessing the data arrays through the domain object leads to page faults within the domain object. This happens despite that CPU and GPU mostly access a different set of data arrays. By diagnosing and resolving this issue we were able to improve the performance up to 3x over the baseline. §IV-A discusses the optimizations in more detail.

III. XPlacer - Design and Implementation

This section introduces three memory access anti-patterns and describes the design and implementation of XPlacer, a source-level instrumentation tool combined with a first runtime system capable of discovering these patterns.

A. Memory access anti-patterns

Hidden data movement cost in heterogeneous codes can often pose a significant obstacle to attaining peak performance. We describe three typical memory access anti-patterns that could significantly degrade algorithm performance.

Alternating CPU/GPU accesses in managed memory: Memory in unified memory space (allocated with `cudaMallocManaged`) is accessible from CPUs and GPUs. The data movement implied from alternating accesses from CPU and GPU is hidden from the programmer and may degrade performance.

Pattern Description:

- Memory allocated with `cudaMallocManaged`.
- Both CPU and GPU access the same allocated memory region.
- Existing `cudaMemAdvise()` data usage hints (*e.g.*, `cudaMemAdviseSetReadMostly`) do not match access characteristics.

Possible remedies:

- Provide appropriate memory access hints for individual memory regions.
- If the accesses are to disjoint regions, the page fault could be caused by an effect similar to false sharing, and splitting an object into a CPU part and GPU part may alleviate the page faults.

Low access density: a GPU only accesses a few memory locations within an allocated memory region. Low access density may lead to algorithms that exhibit low spatial locality, and transfer too much memory per data used.

Pattern Description:

- Memory allocated either with
  - `cudaMalloc` followed by a transfer from CPU to GPU,
  - or `cudaMallocManaged`.
- Only a small fraction of transferred memory gets accessed. Low access density is defined as the number of accessed addresses within an allocated block is lower than a predefined threshold. *accessed* is one if an address was accessed, zero otherwise.

$$\frac{\sum_{addr=start(block)}^{end(block)} accessed(addr)}{size(block)} \leq threshold$$

.

Possible Remedies:

- Partitioned data transfer to overlap computation and communication.
- Data layout optimization to transfer less data.
- Replace `cudaMalloc` with `cudaMallocManaged`.

Unnecessary data transfers:

Pattern Description:

- Memory allocated with `cudaMalloc`.
- The memory is initialized by a transfer from the CPU to GPU.
- The GPU does not access a large chunk of the transferred data, or transfers out a large block of unmodified data.

Possible Remedies:

- Revise the algorithm to eliminate unnecessary transfers of memory that is not accessed or not altered.

B. Instrumentation

XPlacer is developed as a plugin of the ROSE source-to-source translation infrastructure. ROSE supports the analysis of software developed in the C, C++, CUDA, and other programming languages. ROSE represents source code in form of an abstract syntax tree (AST). The ROSE API allows users to manipulate the tree and an unparser converts the modified tree to its corresponding modified source code.

At a high-level, XPlacer replaces possible accesses of heap memory and CUDA function calls with calls to its own tracing functions. The tracing functions mimic the semantic of the operations that they replace and carry out bookkeeping operations that gather data for the runtime analysis.

XPlacer creates instrumented source files that can be compiled with a backend compiler and linked to a runtime system that implements the functions and classes defined in the instrumentation description header file. The resulting executable can be executed on the target system.

Instrumentation details: XPlacer uses an instrumentation description file that defines what code is instrumented. Table I gives a brief overview of XPlacer's instrumentation API. To instrument memory read, write, and read-modify-write (*e.g.*, pre- and postfix increment) expressions, the description file needs to define three functions (*i.e.*, `traceR`, `traceW`, and `traceRW`). The three functions take an address as input and return the same address.

XPlacer instruments any memory read and write that possibly affects memory allocated on the heap. To this end, XPlacer instruments l-valued expressions [13] before they take effect. For example, `*a = 0` becomes `traceW(*a) = 0`. An expression that produces a memory address is instrumented before it is read or written. If the expression context expects an r-value, the expression is instrumented as read, otherwise as write. Any l-value in the context of a read-modify-write operation is instrumented using `traceRW`, for example `traceRW(*a)++`.

To minimize the runtime overhead, the instrumentation is elided when the update does not affect the heap. For example, when variables that have non-reference type are accessed, and when an l-valued expression is used in the context of an expression that does not access the location immediately (*e.g.*, address-of operator, sizeof operator, assignment of a dereferenced pointer to a reference expression).

In addition, the description header file defines functions that replace other functions of interest (*e.g.*, `cudaMalloc`,

TABLE I: Instrumentation API

API	Sample declaration	Description
Memory access tracing The transformation pass uses these functions to instrument read and write accesses to heap memory. They are defined in the instrumentation description file.		
traceR	`template <class T>` `const T& traceR(const T&);`	Used to instrument read operations. `int x = traceR(*p);`
traceW	`template <class T>` `T& traceW(T&);`	Used to instrument write operations. `traceW(*p) = 0;`
traceRW	`template <class T>` `T& traceRW(T&);`	Used to instrument read and update operations.
Function call tracing To trace the effect of functions calls, the transformation pass replaces calls to CUDA functions with tracing functions. Common functions are defined in the XPlacer instrumentation description file. To capture accesses by third party libraries users can provide their own additions.		
xpl replace *funcname*	`#pragma xpl replace cudaMalloc` `cudaError_t` `trcMalloc(void** p, size_t sz);`	replaces calls to *funcname* with calls to the function declaration following the pragma (*i.e.*, `trcMalloc`). `kernel-launch` can be used as *funcname* to replace kernel launches with a call to a custom function.
Diagnostic output Users add these pragmas to the analyzed code to define where the collected data should be analyzed. The transformation pass inserts the actual calls.		
xpl diagnostic *funcname* (*verbatim*; *expanded*)	`#pragma xpl diagnostic \` `trcPrn(std::cout; a)`	Inserts a call to a diagnostic function *funcname*. While *verbatim* arguments are copied as is, pointers to objects in *expanded* arguments are expanded to include all pointers within the referenced object. In addition, expanded arguments are wrapped into a structure XplAllocData, describing the allocation.

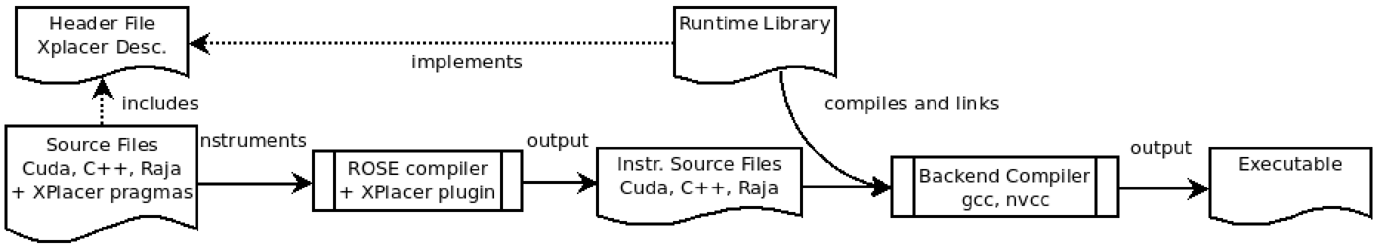


Fig. 1: XPlacer: System Diagram

`cudaMemcpy`, `atomicInc`). Currently, XPlacer can instrument CUDA, C, a significant subset of C++11, and applications based on the RAJA framework.

To analyze an application, programmers would modify the original source code and include the XPlacer header file. In addition, users will need to insert pragmas where they would like the diagnostic module to summarize recent memory accesses. This could be either at the end of a program, after each iteration of the main loop, or after each kernel launch.

The code in Fig. 2 demonstrates how the API for read and write operations is used for instrumenting a pointer access. Since operations on both CPU and GPU are instrumented, the tracer operations (*e.g.*, `traceR`) are marked as `__host__` `__device__`.

```
// interface for reading
template <class T>
__host__ __device__
const T& traceR(const T& el);
// wrapped read operation
int* p = new int(2);
int x = traceR(*p);
```

(a) Read

```
// interface for writing
template <class T>
__host__ __device__
T& traceW(T& el);
// wrapped write operation
int* p = new int(2);
traceW(*p) = 3;
```

(b) Write

Fig. 2: Instrumenting read and write operations

The instrumentation of function calls is controlled through pragmas. Pragmas define which functions in the original code should be replaced by a custom, user-provided function. For example, in order to keep track of all `cudaMallocManaged` calls, a user would add the following pragma.

```
#pragma xpl replace cudaMallocManaged
void traceMallocManaged( void** ptr, size_t sz,
                         unsigned int flags = cudaMemAttachGlobal);
```

The prefix `xpl` is the start of any XPlacer pragma, `replace` indicates that uses of a function with a given name (*i.e.*, `cudaMallocManaged`) shall be replaced by the following function.

To invoke the memory access analysis, XPlacer offers the pragma `diagnostic`. The pragma embeds calls to diagnostic functions in the instrumented code and helps to translate addresses to user level objects.

```
#pragma xpl diagnostic tracePrint(std::cout; a, z)
```

In this example, `tracePrint` is a user-defined function that takes a variable long argument list of user-level object descriptions (*i.e.*, `XplAllocData`). All arguments up to the semicolon are copied verbatim. Any argument after the semicolon (*i.e.*, `a`, `z`) must be a variable that is in scope and that has pointer type. The target types of the pointers are recursively expanded (unless there is type repetition, for example in a linked list) for all pointer members. For every such expression, the XPlacer instrumentation produces a record containing the target address, the access expression (*e.g.*, variable name), and the element size.

Assuming, that `a` points to an STL pair [14] with two integer pointers, and `z` is a pointer to a scalar variable, the embedded diagnostic call would be:

```
tracePrint(std::cout,
           XplAllocData(a, "a", sizeof(*a)),
           XplAllocData(a→first, "a→first", sizeof(*a→first)),
           XplAllocData(a→second, "a→second", sizeof(*a→second)),
           XplAllocData(z, "z", sizeof(*z)));
```

We chose this method that allows users to provide pointers to objects of interest. In our opinion, this approach is more robust than any heuristic that attempts to associate an allocation with a user object. Note, that these user-level objects are only used to give names to allocation records. The tracing and pattern computation would work without them, but the messages would be harder to interpret for users.

Kernel invocations can be wrapped by user defined functions. The replacement is controlled by the pragma `replace kernel-launch`. This method can be used to effectively wrap any kernel execution and to record data before and after the launch of a CUDA kernel, such as the number of page faults reported by the operating system or CUPTI [15]. In the following example, the function `traceKernelLaunch` will replace any kernel launch. `grd`, `blk`, `shmem`, and `stream` are the kernel launch parameters, while `kernel` defines the original kernel function and `Args...` its arguments.

```
#pragma xpl replace kernel−launch
template <class Grid, class Block, class Kernel, class... Args>
__host__
void traceKernelLaunch( Grid grd, Block blk, size_t shmem, cudaStream_t stream,
                        Kernel kernel, Args... args);
```

C. Runtime System

This section discusses a first version of a runtime system that keeps track of relevant information gathered during program execution. The runtime system is designed to discover the patterns described in §III-A. Fig. 3 shows the data layout of XPlacer's runtime system.

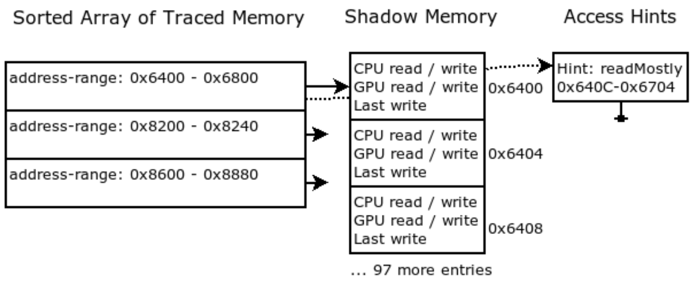


Fig. 3: XPlacer: Shadow Memory

XPlacer wraps calls to memory allocating functions (*e.g.*, `cudaMallocManaged`). For any memory allocation, XPlacer creates an entry in the traced memory table and shadow memory where it stores data about how the memory was accessed. The shadow memory table (SMT) is a sorted data structure that keeps meta information about the allocation. This includes the range of the allocated memory and the allocation function used to allocate the memory. To keep track of actual memory operations, each entry in the SMT has a pointer to the actual shadow memory. Both, the SMT and the actual shadow memory is allocated in unified memory. Since the C-style allocation functions (*e.g.*, `cudaMalloc`) only receive the size of the allocation, but no actual type information, XPlacer uses a shadow entry (a character) for each allocated 32-bit word (roughly a 25% memory overhead.) The wrapper for `cudaFree` deallocates the memory immediately, but delays freeing the shadow memory until the next diagnostic output has been computed.

Whenever a read or write occurs, the entry in the SMT is looked up. If no entry can be found, the address is not tracked, and the memory operation is ignored. If an entry is found, the offset in the shadow memory is computed and the information about this address is updated accordingly. To distinguish between CPU and GPU accesses, XPlacer uses the predefined macro `__CUDA_ARCH__`. The following code snipped shows a sample implementation for tracing reads.

```
// interface for reading
template <class T>
__host__ __device__
const T& traceR(const T& el)
{
  ShadowEntry* traceData = findAllocation(&el);
  if (traceData) {
#if __CUDA_ARCH__
    updateFlagsReadGPU(tracedata);
#else
    updateFlagsReadCPU(tracedata);
#endif
  }
  return el;
}
```

XPlacer uses a bit flag to record the processor of the the last write (either CPU or GPU), whether a CPU or GPU wrote to an address, and whether a CPU or GPU read from an address and whether the data originated on the CPU or GPU. Altogether the runtime system stores seven bits of information (one byte) per 32bit of memory. The diagnostic functions analyze the accesses and identify inefficiencies and access anti-patterns.

In order to invoke the analysis and print out some diagnostics, XPlacer provides a diagnostic function `tracePrint`. Users specify the source location in code using pragma `xpl diagnostic`. For example, in LULESH the diagnostics are called at the end of every timestep. `tracePrint` analyzes the accessed memory regions, prints out messages, and resets the shadow memory.

The collected data supports the identification of the targeted memory access anti-patterns.

Alternating CPU/GPU accesses in managed memory: The shadow memory stores whether the GPU or CPU wrote to a specific memory location (and which processing unit wrote to the location last). In addition, it records any reads from a memory location, and whether the value originated from the CPU or GPU. The runtime analysis examines the recorded data and reports whether there are accesses to the same memory location from both CPU and GPU, where at least one of the accesses is a write.

Low access density: For a user-defined block size, the analysis computes the access density. A memory block is diagnosed if it has at least one access and the computed access density is beneath a specified threshold (*e.g.*, 50%).

Unnecessary data transfers: In order to detect unnecessary data transfers in memory allocated with `cudaMalloc`, we instrument data transfers (*i.e.*, `cudaMemcpy`). Memory transfers from CPU to GPU are recorded as writes by the CPU, while memory transfers from GPU to CPU are recorded as reads by the CPU. The analysis identifies contiguous memory blocks that were transferred to the GPU but never accessed as unnecessary transfers. Similarly, contiguous memory blocks are flagged if they were not updated on the GPU but transferred back to the CPU. The minimum block size of these contiguous memory regions is parametrizable.

D. Finding Performance Issues with XPlacer

To use XPlacer, developers need to perform the following steps: (1) Prepare code for instrumentation. Users modify the source code and include the XPlacer header file. Any code that is placed after the header include directive will be instrumented. (2) Decide where to compute diagnostic messages. To analyze an application, the developer adds one or more pragmas where the traced data will be analyzed and output produced. XPlacer can produce output in the form of a textual summary or in form of raw comma-separated files for further processing (*e.g.*, to produce a graphical output). Typical scenarios range from analyzing each kernel invocation to analyzing an entire application run. Depending on the application, conducting analyses from several perspectives may be useful. (3) Call ROSE with the XPlacer plugin. The ROSE pass produces an instrumented source file. (4) Invoke the backend compiler on the modified file and link with the XPlacer runtime library. (5) Run the application with representative input sets and interpret the results.

Consider LULESH 2 as an example. LULESH uses a singleton object, called the domain. The domain object is allocated in unified memory and contains pointers to arrays (also allocated in unified memory). Most of these pointers are initialized before the first iteration and never modified during execution. However two GPU kernels require temporary storage, thus the CPU allocates unified memory before the kernel execution, sets the domain object's fields, launches the kernel, and frees the temporary storage afterwards. On some architectures such an implementation causes frequent page faults.

To analyze LULESH, a developer would instrument the original code, and analyze the recorded data at the end of each timestep. Fig. 4 shows a sample output produced by XPlacer. The diagnostic output is generated at the end of each timestep iteration. The output is produced for the domain object and all array's that are reachable through it (in total 50 allocations in unified space). The diagnostic output first prints raw access counts of an allocation. `C` and `G` count the number of writes on

```
*** checking 50 named allocations
dom
write counts                      write>read counts
     C         G       C>C       C>G       G>C       G>G
    27         0       680        90         0         0
access density (in %): 9
18 elements with alternating accesses
(dom)->m_p
write counts                      write>read counts
     C         G       C>C       C>G       G>C       G>G
     0      1024         0         0         0      1024
access density (in %): 100
0 elements with alternating accesses
[48 more entries omitted]
```

Fig. 4: LULESH 2: partial XPlacer output after the second iteration

CPU and GPU, respectively. Note, multiple writes to the same address by the same device are counted as one. The next four columns from `C>C` to `G>G` indicate the number of addresses that have been read (in each category an address is counted at most once). The read counts are further subdivided by a value's origin (the location of the preceding write). Thus `C>C` refers to reads by the CPU of values previously written by the CPU. Similarly, `C>G` indicates a read by the GPU that was preceded by a write on the CPU. The preceding write is the last write to that address regardless if it occurred in the same iteration or earlier (*e.g.*, at start up).

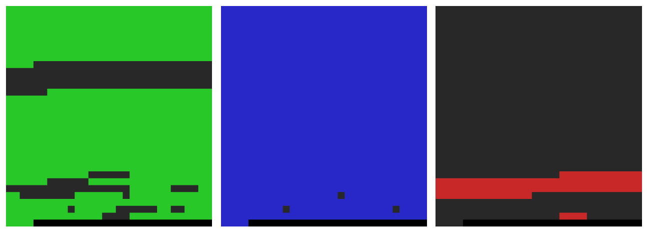
The next two lines are produced by the anti-pattern analysis for one iteration: Low access-density and alternating CPU/GPU accesses.

The sample output shows the results for the domain object (`dom`) and one array reachable through `(dom)->m_p` after the second iteration. Other elements are omitted.

A developer would analyze the recorded data at the end of each time timestep. The analysis of the first iteration indicates that the GPU utilizes data initialized by the CPU. The analysis of later iterations indicate that almost all user-level objects are accessed exclusively either by the CPU or the GPU. The exception is the domain object, which is accessed by both the CPU (read and write) and the GPU (read).

To illustrate this further, Fig. 5 shows a graphical representation of CPU and GPU accessing the domain object. Since the GPU does not update the domain object, we do not show the maps related to GPU writes (they would be empty). The first iteration includes program initialization. After the first time-step the domain object is rarely modified. The exception is the temporary storage fields that are set by the CPU and accessed by the GPU. Figures 5e and 5f show where the GPU accesses overlap with CPU writes.

A developer would look at the output to find red flags, such as `dom`'s 18 elements with alternating accesses. A run with a profiler will confirm that this translates to numerous page-faults if `cudaMemAdviseSetReadMostly` is not set. To mitigate the problem, the developer will need to choose one of several strategies discussed in §IV-A.


(a) CPU reads CPU (b) CPU writes (c) GPU reads CPU

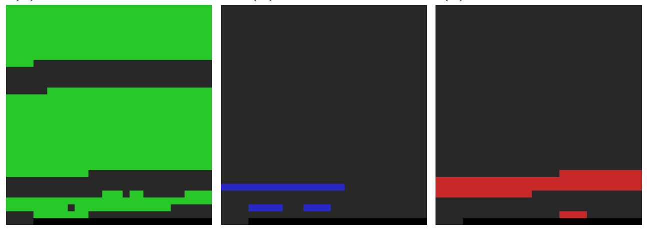

(d) CPU reads CPU (e) CPU writes (f) GPU reads CPU

Fig. 5: LULESH 2: Access maps of the domain object (the domain object has a size of 3736 bytes). 5a, 5b, and 5c show maps of all memory locations accessed during initialization and the first iteration. 5d, 5e, and 5f show similar maps for the second and later iterations.

IV. Evaluation

XPlacer has been evaluated on several sample applications. Unless otherwise noted, all code was compiled with -O3 (using CUDA 9.2.88 with gcc 7.3) and run on an Intel E5-2695 v4 (2.1Ghz) with an Nvidia Pascal CPU and a IBM Power 9 system (2.3Ghz) with an Nvidia Volta GPU. On the IBM system, the CPU and GPU were connected via Nvlink [16].

A. *LULESH*

The first applications is LULESH 2, an HPC proxy application for hydrodynamics codes. The tested LULESH version is based on the RAJA framework and configured for utilizing CUDA as the kernel execution strategy and for running on a single node (without MPI). In the default RAJA/CUDA version, LULESH allocates dynamic memory in managed space without any further hints.

§III-D discussed how XPlacer can be used to find why the application may page fault on some devices. In order to remedy the problem, we experimented with different solutions: (1) Provide hints to the CUDA runtime for better management of unified memory. We tried `cudaMemAdviseSetReadMostly`, `cudaMemAdviseSetPreferredLocation` to CPU, `cudaMemAdviseSetAccessedBy` the GPU and CPU. (2) Duplicate the domain object, so that an object is exclusively accessed by either the CPU or the GPU, and pass the pointers to temporary storage outside the domain object.

We timed LULESH for problem sizes between 8 to 48. The performance graphs in Fig. 6 show the obtained performance. In addition to the two described systems, we also run tests on an Intel system (E5-2698 v3) with a Volta GPU (Cuda version 9.1.85). On the Intel systems without Nvlink, we saw

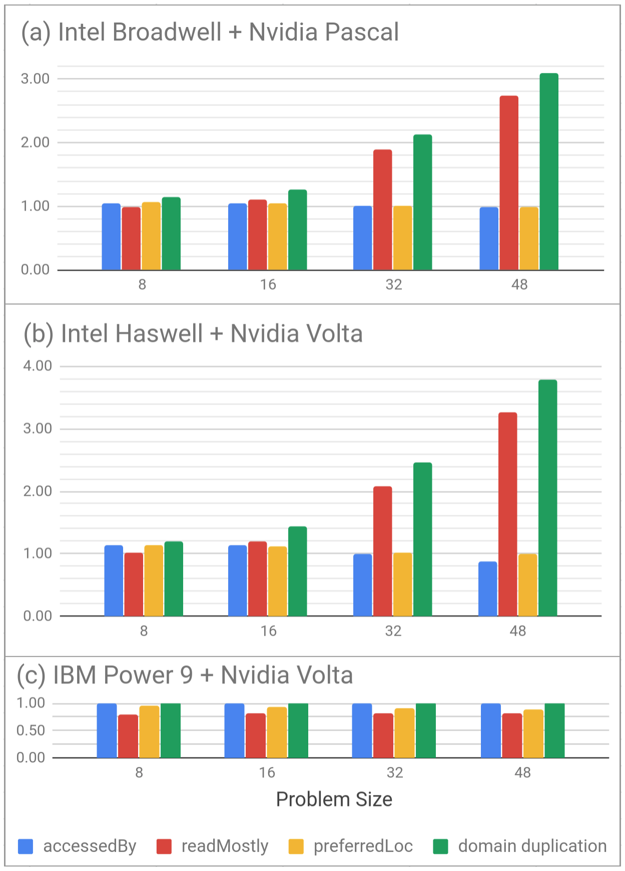


Fig. 6: LULESH 2: Speedup over the baseline. Four different methods were used to remedy a large number of CPU page faults in the default implementation. The experiments were performed for three combinations of CPU/GPU on four different problem sizes ranging from 8 to 48. The execution of the baseline code took 1.4s, 4.4s, 18.3s, and 47.3s on Pascal, 1.4s, 4.6s, 19.6s, and 52.2s on Intel plus Volta, and 1.8s, 4.9s, 11.7s, and 20s on IBM plus Volta.

a performance speedup of 2.75x (Intel/Pascal) and 3.1x (Intel/Volta) for large data sizes by simply setting the memory advise `cudaMemAdviseSetReadMostly` (one line change of the source). The largest performance improvement of 3.1x (Intel/Pascal) and 3.7x (Intel/Volta) were obtained by creating two identical domain objects where each was exclusively accessed by either CPU or GPU. In contrast, on the IBM/Volta system object duplication yielded a marginal speedup of 1.03x, whereas memory hints did not improve performance. Setting `cudaMemAdviseSetReadMostly` resulted in a speedup of 0.8x (20% slower than the original code). We attribute the performance difference between the tested systems to the availability of Nvlink on the IBM Power architecture.

B. *Smith-Waterman*

The second algorithm that we examined was an implementation of Smith-Waterman [17]. Smith-Waterman is an algorithm that computes the longest common subsequence of

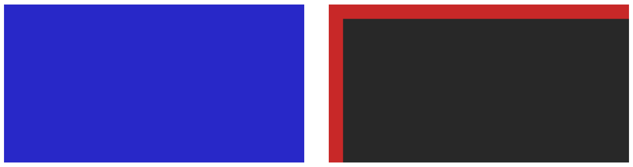

(a) CPU writes (b) GPU reads CPU

Fig. 7: Smith-Waterman, input size = 20x10; 7a shows the CPU initializes the entire H matrix, but 7b shows that only boundary values are accessed.

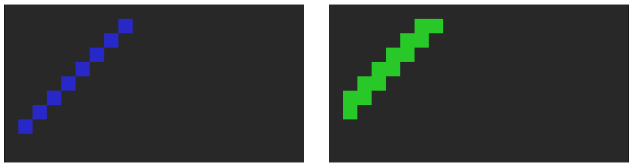

(a) GPU writes (b) GPU reads GPU

Fig. 8: Smith-Waterman, input size = 20x10; GPU accesses to the H matrix in Iteration 8. 8a shows the values written by the GPU, and 8b shows values that were accessed (and produced by the GPU in the previous two iterations). For large data sets, the accesses along the diagonals become prone to page-faults.

two molecular strings. The algorithm takes two input strings (*e.g.*, `a` and `b`) of length n and m and produces two $n\ x\ m$ matrices `H` and `P` that store the best score and path, respectively. The examined implementation allocates storage for the four data elements using `cudaMallocManaged`, transfers `a` and `b` from the original storage, and zeroes out the matrices. We instrumented the code using XPlacer and first ran the analysis at the end of the algorithm. This revealed that the initial zero values of most matrix elements were never accessed. Only the boundary zeroes were used by the algorithm. Their initialization could be performed on the fly. Fig. 7 shows a graphical representation of the analysis result.

The second analysis was conducted after each iteration. Fig. 8 depicts the GPU writes and GPU reads from earlier GPU writes. The analysis reveals low access density. Only three memory locations that are contiguous in memory are accessed in each iteration. Data sets exceeding the amount of GPU memory will cause a significant number of page faults.

Optimizing Smith-Waterman: We used this information to modify the Smith-Waterman algorithm in two ways. (1) We initialized the boundary values of the `H` matrix on the fly. This did not produce any speedup. (2) We rotated the matrix by 45 degrees so that each iteration only accesses contiguous memory within the matrix. The modified algorithm runs significantly faster. Fig. 9 shows the achieved speedup for varied input sizes on the test systems. On the Intel plus Pascal system, the memory advise `setPreferredLocation` to GPU was used for all unified memory allocations; on the IBM plus Volta system, this advise was not set, because it caused performance degradation for the largest input size. On the IBM plus Volta system, running the optimized algorithm with 46000 character

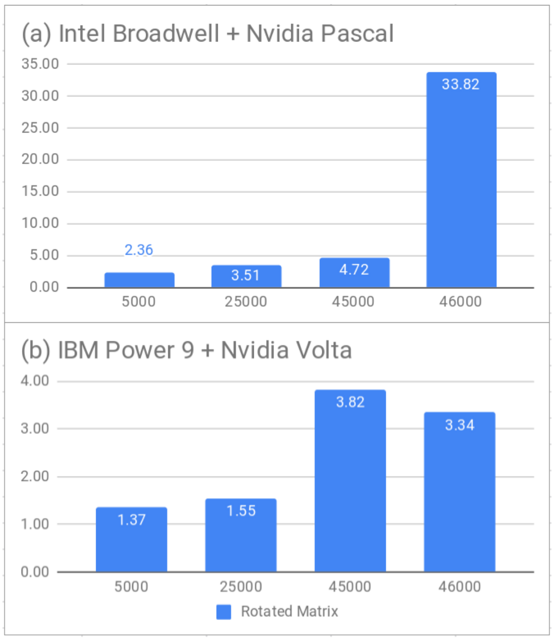


Fig. 9: Smith-Waterman: Speedup over the baseline. The modified version uses a rotated matrix to reduce the resident memory size on a GPU. The input length for both strings were 5000, 25000, 45000 (data sets fits in GPU memory), and 46000 (data set size exceeds GPU memory). The baseline code executed in 78ms, 863ms, 3034ms, and 24928ms on Pascal (with `setPreferredLocation` to GPU), and in 86ms, 927ms, 6455ms, and 45108ms on Volta (without `setPreferredLocation`).

long input strings exhibited significant performance variability in the range from 2300ms to 17000ms. Profiling the optimized algorithm with 45000 and 46000 character long strings shows that the relatively slow performance of a typical run can be attributed to roughly 12s spent on "GPU page fault groups". Further optimization of this algorithm is ongoing work.

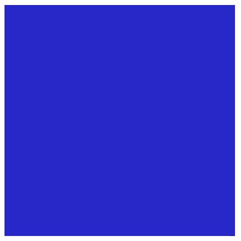
C. Rodinia benchmarks

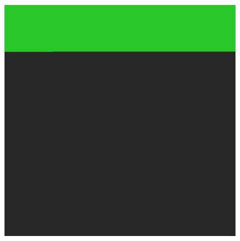
We also analyzed several Rodinia CUDA benchmarks, Backprop, CFD, Gaussian, LUD, NN, and Pathfinder. Where applicable, we ran the analysis after each iteration and at the end of the program. Table II summarizes our findings.

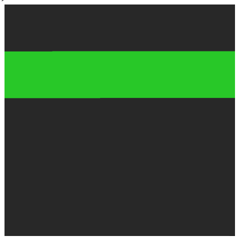
Optimizing Pathfinder: Pathfinder exhibits a regular access pattern as depicted in Fig. 10. Based on these data, we conjectured that overlapping data transfer with computation would be beneficial. Instead of transferring `gpuWall` as a whole, the modified code only transfers the array section that is going to be accessed by the kernel. The Pathfinder benchmark takes three parameters as input, number of columns, number of rows (r), and pyramid size (p). Each kernel processes $\frac{r}{p}$ rows and accesses $\frac{100p}{r}$ percent of the `gpuWall`. For small data sizes, the modified code runs slower. For medium sizes, Fig. 11 shows that the revised version runs up to 1.13x faster on an Intel plus Nvidia Pascal system. For the same inputs, the revised version remains slower on IBM plus Nvidia Volta.

TABLE II: Findings in a subset of the Rodinia benchmarks

Benchmark	Findings
Backprop	An array `output_hidden_cuda` is allocated but never used. An array `input_cuda` is copied from CPU to GPU and then back to CPU, although it is not modified by the GPU.
CFD	no possible improvements identified.
Gaussian	An array `m_cuda` is allocated on the CPU and transferred to the GPU. The GPU overwrites all values transferred from the CPU before they are used. Thus, the initial data transfer can be eliminated.
LUD	A two-dimensional array `m_d` is initialized on the CPU, transferred to the GPU, where it is recomputed and transferred back to CPU. The analysis revealed that the first row is never updated. The GPU computation accesses most memory allocations during the first iteration. As the computation progresses fewer and fewer memory locations are accessed on the GPU. This may create some opportunities to transfer out the data from the GPU to CPU early (while the GPU is busy with later iterations).
NN	no possible improvements identified.
Pathfinder	An array `gpuWall` is produced on the CPU and is bulk transferred to GPU before the computation begins. With N being the number iterations, only $\frac{100}{N}\%$ of the data is accessed in each iteration.


(a) CPU writes - Iteration 1


(b) GPU reads CPU - Iteration 2


(c) GPU reads CPU - Iteration 2

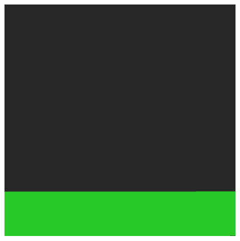

(d) GPU reads CPU - Iteration 5

Fig. 10: Rodinia pathfinder benchmarks: the access maps show the array `gpuWall`. It is initialized by the CPU and copied to the GPU (a). Each iteration accesses a fifth of the access space. This is shown for the first (b), second (c), and fifth (d) iteration.

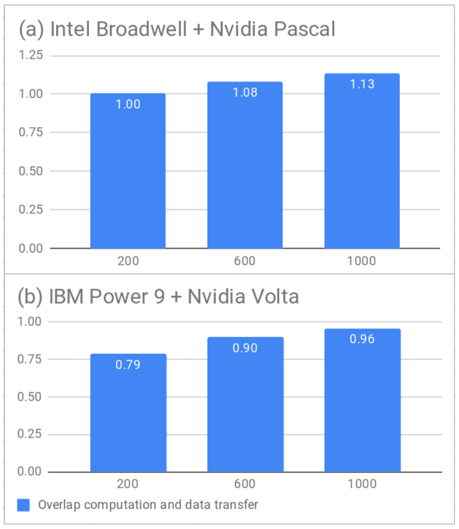


Fig. 11: Pathfinder: Speedup over the baseline. The modified version overlaps computation with the transfer of `gpuWall`'s segment required by the next iteration. The experiments were performed for two combinations of CPU/GPU. The input sizes were 1M columns, varied size of rows (200, 600, 1000), and a pyramid height of 20. The baseline code had an average runtime of 434ms, 891ms, 1488ms on Pascal, and of 214ms, 417ms, and 598ms on Volta.

We also eliminated unnecessary data transfer and allocation in the benchmarks backprop and gaussian. These improvements did not produce a significant speedup over the baseline.

D. Runtime Overhead

The performance overhead incurred by XPlacer can be traced back to a few factors. The shadow memory table is a sorted array of memory addresses. *Allocating and deallocating* requires an update of the table – an O(N) operation. The tested benchmarks mostly allocate memory at startup time and deallocate all memory at the end. For every potential *access to heap memory*, the shadow memory table is searched for a corresponding entry. Lookup of an entry uses linear search when the number of allocations is less than 64, and binary search otherwise. With more than 50 dynamically allocated memory regions, LULESH is the benchmark with the most allocations. Another source of inefficiencies are *potential side-effects of instrumentation functions*. These prevent the compiler from performing many aggressive optimizations. *Accessing shadow memory* increases the amount of memory that the GPU needs to keep resident. This leads to an increase in data transfer or inability to run an instrumented application if memory resources are scarce on the GPU.

Table III shows the performance impact of XPlacer's instrumentation on a number of benchmarks. The timing excludes overhead for printing the diagnostic output messages. On average, the instrumented runtime is about 15x slower than the original version.

V. Related Work

The work most similar to XPlacer is CudaAdvisor [18]. CudaAdvisor is a profiling framework built on top of the LLVM toolchain [19]. CudaAdvisor is implemented as LLVM pass that instruments both host and device bitcode. CudaAdvisor tracks calls and returns (to get a full call stack), memory allocations and deallocations, plus data transfers between host and device. Depending on the specific need, CudaAdvisor may also instrument memory and arithmetic operations, as well as control flow choices (*e.g.*, to detect warp divergence [20]). In contrast to CudaAdvisor, XPlacer instruments source code. On one side, instrumented binary code typically runs faster because the instrumentation can be added to an optimized binary, whereas source-level instrumentation often poses an obstacle to compile-time code optimization. On the other side, instrumenting source code offers better portability across CUDA versions and is easier to understand for programmers not familiar with low-level details. XPlacer's function level instrumentation can be more easily customized through pragmas.

SASSI [21] is a binary instrumentation toolkit. Users can inject code at user-defined locations, including control flow transfers, memory operations, and kernel entry and exit, in order to gather runtime data. SASSI extends the Nvidia backend and could also be run as just-in-time compiler. SASSI can pass memory addresses and register information to user-defined functions. SASSI has been used to identify control flow divergence, memory divergence, warp-level value analysis, and error injection.

The TAU toolkit [22], [23] is a profiling framework for HPC systems. TAU supports profiling CUDA kernels by instrumenting functions, blocks, and statements. TAU may replace system calls with their own implementation to gather performance data. TAU can also make use of Nvidia's CUPTI low-level profiling interface. TAU gathers performance information of GPU computations and integrates it with other application performance data, through instrumentation of functions, methods, basic blocks, and statements to capture a performance picture of the resulting application execution.

StructSlim [24] is a light-weight profiler for binary code. StructSlim samples memory accesses and identifies access patterns where structure splitting could optimize memory latencies.

CUPTI [15] is Nvidia low-level profiling toolkit. CUPTI offers APIs for profiling, metrics, and event handling on the GPU. CUPTI is the interface of choice for other toolkits, such as TAU and SASSI. CUPTI is complementary to our tool. XPlacer could be used to insert CUPTI calls before and after kernel calls in order to collect interesting events. For example, we could collect page fault events in order to more precisely pinpoint their locations.

RTHMS [25] is a tool that automatically places objects in memory. To this end, RHTMS analyzes a program and based on a set of rules decides to what kind of memory objects should be assigned.

TABLE III: Runtime overhead (Intel + Pascal)

Benchmark	**Configuration**	**Overhead**
LULESH 2	size = 8, iterations = 16	14x
LULESH 2	size = 48, iterations = 16	15x
LULESH 2	size = 96, iterations = 16	18x
Smith-Waterman	size = 1000x1000	20x
Smith-Waterman	size = 10000x10000	13x
Smith-Waterman	size = 20000x20000	8x
Backprop	size = 640K	5x
Gaussian	size = 100, kernel execution time	14x
Gaussian	size = 1000, kernel execution time	12x

Caliper [26] is a toolkit that simplifies the collection of performance data from applications that are composed from various software libraries. Caliper combines data collected from different parts in form of user definable key-value pairs.

Diogenes [27] is a GPU tool that identifies inefficiencies in CPU/GPU code, such as unnecessary synchronization and memory transfers. Diogenes uses feed-forward measurements that optimize the instrumentation over multiple runs.

HPCToolKit [28] is a tool that uses statistical sampling of timers and hardware counters to gather data from running applications. HPCToolkit analyzes binaries and is capable of analyzing heterogeneous programming models [29].

VI. Conclusion and Future Work

This paper has introduced the XPlacer framework and demonstrated how to utilize it for gaining a better understanding of data transfers and data accesses in heterogeneous applications. The gathered data was used to identify a performance bottleneck in LULESH, a widely used HPC proxy application, and inefficiencies in other test codes.

Currently, the runtime system tracks how data is accessed and matches that against three memory access anti-patterns. The presented runtime is a prototype library. XPlacer can be used to introduce a wide set of instrumentations. To more precisely pinpoint when and how memory accesses cause performance degradation, a runtime could more precisely model the GPU memory hierarchy and data transfer rules, or use CUPTI or operating system level performance counters.

Acknowledgments

We would like to thank the anonymous referees for their helpful suggestions for improving this manuscript.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and supported by LLNL-LDRD 18-ERD-006. LLNL-CONF-795057.

References

[1] P. Kharya, "Record 136 NVIDIA GPU-accelerated supercomputers feature in TOP500 ranking," November 2019, accessed on January 23, 2019. [Online]. Available: https://blogs.nvidia.com/blog/2019/11/19/record-gpu-accelerated-supercomputers-top500/

[2] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[3] ISO/IEC 14882:2017(E) International Standard, *Programming Language C++*. JTC1/SC22/WG21 - The C++ Standards Committee, 2017.

[4] M. Bari, L. Stoltzfus, P.-H. Lin, C. Liao, M. Emani, and B. Chapman, "Is data placement optimization still relevant on newer GPUs?" *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 83–96, 2018.

[5] R. Landaverde, Tiansheng Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2014, pp. 1–6.

[6] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, 2011.

[7] The ROSE compiler group at the Lawrence Livermore National Laboratory, “The ROSE source-to-source compiler,” 2019, accessed on June 19, 2019. [Online]. Available: http://rosecompiler.org

[8] I. Karlin, J. Keasler, and R. Neely, “Lulesh 2.0 updates and changes,” Tech. Rep. LLNL-TR-641973, August 2013.

[9] ——, “Raja proxy applications,” 2017, accessed on September 3, 2019. [Online]. Available: https://github.com/LLNL/RAJAProxies/tree/master/lulesh-v2.0/RAJA

[10] M. Harris, “Unified memory for CUDA beginners,” June 2017, accessed on July 23, 2019. [Online]. Available: https://devblogs.nvidia.com/unified-memory-cuda-beginners/

[11] Nvidia Corp., “Nvidia CUDA runtime API,” May 2019, accessed on July 23, 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[12] R. Hornung, H. Jones, J. Keasler, R. Neely, O. Pearce, S. Hammond, C. Trott, P. Lin, C. Vaughan, J. Cook, R. Hoekstra, B. Bergen, J. Payne, and G. Womeldorff., “ASC tri-lab co-design level 2 milestone report 2015,” The Larewnce Livermore National Laboratory, Tech. Rep., Sep. 2015, lLNL-TR-677453.

[13] J. C. Mitchell and K. Apt, *Concepts in Programming Languages*. USA: Cambridge University Press, 2001.

[14] D. R. Musser, G. J. Derge, and A. Saini, *STL tutorial and reference guide, second edition: C++ programming with the standard template library*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[15] Nvidia Corp., “CUDA profiling tools interface,” Aug. 2019, accessed on August 30, 2019. [Online]. Available: https://docs.nvidia.com/cupti/Cupti/index.html

[16] IBM POWER9 NPU team, “Functionality and performance of NVLink with IBM POWER9 processors,” *IBM Journal research & development*, vol. 62, no. 4/5 paper 9, July/September 2018. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8392669

[17] T. Smith and M. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022283681900875

[18] D. Shen, S. L. Song, A. Li, and X. Liu, “CudaAdvisor: LLVM-based runtime profiling for modern GPUs,” in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, 2018, pp. 214–227. [Online]. Available: http://doi.acm.org/10.1145/3168831

[19] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[20] J. Anantpur and R. Govindarajan, “Taming warp divergence,” in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO ’17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 50–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049832.3049839

[21] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O’Connor, and S. W. Keckler, “Flexible software profiling of GPU architectures,” *SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 185–197, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2872887.2750375

[22] S. S. Shende and A. D. Malony, “The tau parallel performance system,” *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 2006. [Online]. Available: http://dx.doi.org/10.1177/1094342006064482

[23] Nvidia Corp., “TAU performance system,” 2019, accessed on September 3, 2019. [Online]. Available: https://developer.nvidia.com/tau-performance-system

[24] P. Roy and X. Liu, “StructSlim: A lightweight profiler to guide structure splitting,” in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO ’16. New York, NY, USA: ACM, 2016, pp. 36–46. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854053

[25] 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 *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2017. New York, NY, USA: ACM, 2017, pp. 82–91. [Online]. Available: http://doi.acm.org/10.1145/3092255.3092273

[26] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, “Caliper: Performance introspection for hpc software stacks,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 47:1–47:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014967

[27] B. Welton and B. Miller, “Diogenes: Looking for an honest CPU/GPU performance measurement tool,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’19, 2019. [Online]. Available: ftp://ftp.cs.wisc.edu/paradyn/papers/welton_diogenes.pdf

[28] N. R. Tallent and J. M. Mellor-Crummey, “Effective performance measurement and analysis of multithreaded applications,” in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’09. New York, NY, USA: ACM, 2009, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504210

[29] J. Mellor-Crummey, “Extending HPCToolkit for GPU-accelerated systems,” Sep. 2018, presentation at the Scalable Tools Workshop.