Efficient Mapping of Irregular C++ Applications to Integrated GPUs

Rajkishore Barik
Intel Labs
Santa Clara, CA
rajkishore.barik@intel.com

Rashid Kaleem
University of Texas
Austin, TX
rashid@cs.utexas.edu

Deepak Majeti
Rice University
Houston, TX
dm14@rice.edu

Brian T. Lewis
Intel Labs
Santa Clara, CA
brian.t.lewis@intel.com

Tatiana Shpeisman
Intel Labs
Santa Clara, CA
tatiana.shpeisman@intel.com

Chunling Hu
Intel Labs
Santa Clara, CA
chunling.hu@intel.com

Yang Ni
Google Inc.
Mountain View, CA
yang.ni@gmail.com

Ali-Reza Adl-Tabatabai
Google Inc.
Mountain View, CA
alirezaadltabatabai@gmail.com

ABSTRACT

There is growing interest in using GPUs to accelerate general-purpose computation since they offer the potential of massive parallelism with reduced energy consumption. This interest has been encouraged by the ubiquity of integrated processors that combine a GPU and CPU on the same die, lowering the cost of offloading work to the GPU. However, while the majority of effort has focused on GPU acceleration of regular applications, relatively little is known about the behavior of irregular applications on GPUs. These applications are expected to perform poorly on GPUs without major software engineering effort. We present a compiler framework with support for C++ features that enables GPU acceleration of a wide range of C++ applications with minimal changes. This framework, Concord, includes a low-cost, software SVM implementation that permits seamless sharing of pointer-containing data structures between the CPU and GPU. It also includes compiler optimizations to improve irregular application performance on GPUs. Using Concord, we ran nine irregular C++ programs on two computer systems containing Intel 4^{th} Generation Core processors. One system is an Ultrabook with an integrated HD Graphics 5000 GPU, and the other system is a desktop with an integrated HD Graphics 4600 GPU. The nine applications are pointer-intensive and operate on irregular data structures such as trees and graphs; they include face detection, BTree, single-source shortest path, soft-body physics simulation, and breadth-first search. Our results show that Concord acceleration using the GPU improves energy efficiency by up to 6.04× on the Ultrabook and 3.52× on the desktop over multicore-CPU execution.

CGO '14, February 15-19 2014, Orlando, FL, USA
 ACM 978-1-4503-2670-4/14/02 $15.00.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*; D.3.4 [**Programming Languages**]: Processors—*compilers, code generation*

General Terms

Algorithms, Performance, Experimentation

Keywords

Integrated GPU programming, Compiler optimization, Energy efficiency

1. INTRODUCTION

Graphics processor units (GPUs) have become increasingly popular to accelerate general-purpose computation. GPUs provide massive parallelism on a small energy budget and offer opportunities for significant energy savings and performance improvements compared to multicore CPUs. Interest in GPU acceleration is also fueled by the ubiquity of *integrated* processors from hardware vendors such as Intel and AMD. These processors integrate a CPU and GPU onto the same die where they share resources like physical memory (and on Intel's integrated processors, the last-level cache). The advantage of integrated GPUs is that they benefit from low-latency communication and eliminate most data copying, which significantly lowers the cost of offloading work to the GPU. However, integrated GPUs are limited by the power and size budget allocated for the integrated processor.

This interest in GPU acceleration of applications has led to the development of specialized programming languages such as CUDA [6], OpenCL [4], and, more recently, OpenACC [12] and Microsoft C++ AMP [5]. These specialized languages expose details of the GPU architecture and CPU/GPU communication model to the programmer. While

they enable expert programmers to achieve high performance, their complexity and the architectural understanding they require limit widespread use of GPU programming.

One way to reduce the complexity of GPU programming is to use the same data-parallel programming models that are already used for programming multi-core CPUs. Recent work [19, 14] demonstrated the practicality of this approach for regular applications operating on array-based data structures. The question, though, remains whether benefits of GPU execution can be extended to irregular applications written in an object-oriented programming style that features object references, virtual functions, and functor-based parallel constructs.

This paper describes Concord, a heterogeneous C++ programming framework for processors with integrated GPUs designed to allow general-purpose, object-oriented, data-parallel programs to take advantage of GPU execution. Concord supports most C++ features, including namespaces, templates, multiple inheritance, operator and function overloading, as well as virtual functions. It supports two parallel constructs for offloading computation to the GPU: a parallel-for loop and a parallel-reduction loop. These constructs are modeled after ones provided by Thread Building Blocks (TBB) [3], and are similar to ones provided by other CPU parallelism frameworks such as OpenMP, TPL [10], and Cilk [11]. Most importantly, Concord supports seamless sharing of data between the CPU and GPU via an efficient software implementation of shared virtual memory (SVM) augmented with compiler optimizations to reduce the overhead of shared pointer translations.

SVM enables heterogeneous programs to directly share pointer-containing data structures between the CPU and GPU. Since object-oriented programs make heavy use of objects that point to other objects, SVM is a prerequisite for GPU execution of object-oriented C++ programs. CUDA supports pointer sharing between the CPU and GPU. However, CUDA's SVM requires hardware support and is limited to NVIDIA's Fermi-class GPUs. In contrast, our SVM solution is implemented purely in software and targets integrated GPUs with no memory consistency between CPU and GPU such as processors readily available today from Intel and AMD. As maintaining CPU-GPU consistency increases processor complexity and die area, it is unrealistic to expect all GPUs, including low-end GPUs for mobile devices, to support SVM in hardware.

We evaluate Concord using nine realistic irregular C++ applications running on two computer systems with Intel 4^{th} Generation Core processors. These applications are pointer-intensive as they operate on irregular data structures (trees and graphs) represented in the traditional C/C++ fashion using pointers. Many are quite large. For example, ClothPhysics [8], developed by Intel to illustrate capabilities of multi-core CPUs, has more than 9000 lines of C++ code including 400 lines of code in parallel_for and parallel_reduce loops that we target for offloading to the GPU. Our results indicate that GPU execution can bring significant benefit to these applications resulting in energy savings of up to 6.04× and 3.52× on these two integrated systems.

To summarize, we make the following contributions:

- We demonstrate that C++ applications that use pointers and virtual functions can be automatically and transparently mapped to the GPU.
- We show that shared virtual memory (SVM) can be successfully implemented in software on off-the-shelf systems with no specialized hardware support for SVM.
- We describe novel compiler optimizations that reduce the overhead of software-based SVM implementation and cache contention among the GPU cores.
- We show that GPU execution can bring significant energy benefits to irregular applications even without sophisticated data restructuring or algorithmic changes. We ran nine realistic irregular programs on two computer systems with integrated Intel 4^{th} Generation Core processors. Our results demonstrate an average 2.04× and 1.69× energy savings over execution on a multi-core CPU.

The rest of the paper is organized as follows. Section 2 presents the Concord programming language constructs and restrictions. Section 3 then describes the details of our prototype implementation. Section 4 describes compiler optimizations designed to improve the performance and energy efficiency of generated GPU code. Sections 5 provides experimental results. Section 6 discusses related work and Section 7 concludes.

2. PROGRAMMING MODEL

Concord supports most C++ features with some exceptions. It provides two API functions for data-parallel iteration and reduction and has SVM support that enables programs to transparently share pointer-containing data structures.

2.1 Support for C++

Concord supports most C++ features in the GPU code including classes, virtual functions, multiple inheritance, operator and function overloading, templates, and namespaces. However, due to compiler and GPU hardware limitations, there are restrictions to its C++ support, violations of which result in compile-time warnings and `parallel_for_hetero` or `parallel_reduce_hetero` code being executed on the CPU. In particular, Concord does not support recursion (except for tail-recursion that can be eliminated at the compile time), function calls via a function pointer, taking the address of a local variable, memory allocation on GPU, and exceptions. We plan to lift the last two restrictions as part of the future work. Note that although Concord does not support function calls via a function pointer, it supports virtual and externally defined functions.

2.2 Concord programming constructs

Concord's two template API functions for data-parallel computation are modeled after the corresponding ones in Intel Threading Building Blocks (TBB).

```
template <class Body>
void parallel_for_hetero(int n, const Body &b,
    bool on_CPU);

template <class Body>
void parallel_reduce_hetero(int n, const Body &b,
    bool on_CPU);
```

```
1  class LoopBody {
2    Node * nodes;  // array of nodes
3   public:
4    LoopBody(Node *arr) : nodes(arr) {}
5    void operator()(int i) { // executed in parallel
6      nodes[i].next = &(nodes[i+1]);
7    }
8  };
9
10 void convertToLinkedList(Node * array, int N) {
11    LoopBody *b = new LoopBody(array);
12    parallel_for_hetero(N, *b, FALSE);
13 }
```

```
1  typedef unsigned long CpuPtr;
2  #define AS_GPU_PTR(T,p) (__global T *)(&svm_const[(p)])
3
4  __kernel void operator_1(__global char *gpu_base,
5           CpuPtr cpu_base, CpuPtr cpu_ptr) {
6    uint i = get_global_id(0);
7    __global char *svm_const = (gpu_base - cpu_base);
8
9    __global Node *gpu_ptr = AS_GPU_PTR(Node, cpu_ptr);
10
11   *(AS_GPU_PTR(Node, gpu_ptr[i].next)) =
12         &gpu_ptr[i+1];
13 }
```

Figure 1: Left hand side shows a Concord program that converts an array of *Node* objects to a linked list in parallel. The OpenCL code generated by our compiler for the `operator()` function is shown on the right.

Both template functions take a parameter `n` that specifies the iteration space, $[0..n)$ to be done in parallel. For both functions, the second parameter `b` must be an instance of a class `Body` that defines a function call operator `operator ()` specifying the body of the parallel loop or reduction. The third parameter controls whether execution should be on the CPU or GPU. For `parallel_reduce_hetero`, the `Body` class must define an additional method `join` to combine the results for two `Body` objects.

Concord does not guarantee that the different loop iterations will be executed in parallel. Also, as in TBB, programmers should make no assumption about the order in which different iterations are done. Similarly, floating point determinism in reductions is not guaranteed.

An example showing the use of `parallel_for_hetero` appears in the left hand side of Figure 1. This example illustrates how it might be used to convert an array of pointers to a singly-linked list data structure in parallel.

2.3 Shared virtual memory support

To make it as easy as possible to run existing C++ programs on an integrated processor, Concord provides SVM. This allows programs running on the CPU and GPU to directly share complex, pointer-containing data structures such as trees and linked lists. SVM also eliminates the need to marshal data between the CPU and GPU.

Concord maintains the consistency of memory before and after offloading work. It ensures that all updates made by the CPU are visible to the GPU at the start of executing a `parallel_for_hetero` and `parallel_reduce_hetero` call. After finishing their execution, Concord also ensures that any GPU updates are visible to the CPU.

3. IMPLEMENTATION

Figure 2 depicts the components of our Concord framework along with their interaction with other components. We use the CLANG and LLVM infrastructure to compile Concord C++ programs. A compiler pass identifies the heterogeneous loop body functions (*i.e.,* the `operator()` and `join` methods of a body class) and generates CPU code as well as GPU OpenCL kernel code for them. We generate a host-side executable that embeds the generated OpenCL. Later, to execute a heterogeneous loop, the runtime extracts its OpenCL code, just-in-time compiles it to GPU ISA if necessary via the vendor-specific OpenCL compiler, and then, based on the `on_CPU` flag, decides whether to execute it on the CPU or GPU. The compiler and runtime details are described below.

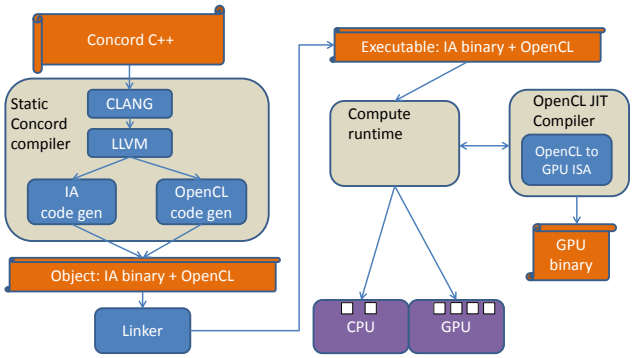

Figure 2: Overall flow diagram of the Concord framework

3.1 CPU-GPU shared pointers

Concord's SVM support allows the GPU to share the same pointers as the CPU. Concord represents a shared pointer using the CPU virtual memory address, and Concord's compiler generates code to translate virtual addresses on the GPU at runtime. The challenge of implementing this translation is that the CPU and GPU may have separate virtual-to-physical mappings and different pointer representations. These details differ greatly from one processor architecture to the next. The remainder of this section describes our implementation on Intel's 4^{th} Generation Core processor.

On this processor, the GPU and CPU use separate page tables. The GPU's virtual address space is segmented into *surfaces* and each surface is referenced by a binding table entry. A GPU pointer is represented as a binding table index plus an offset. To access memory, the offset is added to the surface's base address obtained by looking up that surface's binding table entry. Thus, when we dereference a shared pointer on the GPU, we must translate that CPU virtual address so that it refers to the same physical memory location on both GPU and CPU.

To do this translation, we create a virtual memory re-

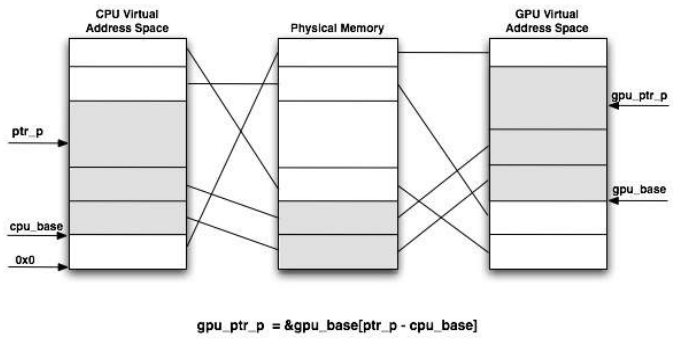

Figure 3: CPU and GPU shared pointer transformation

gion at program startup that is shared between the CPU and GPU[1]. Any shared pointer that the GPU needs to dereference must be allocated in this shared memory region. We achieve this by redirecting malloc and free to specialized routines that allocate and free memory in the shared memory region. The shared memory region is pinned during GPU kernel execution and has a backing GPU surface with a binding table entry that is constant during runtime. This approach substantially reduces the cost of Concord's shared pointer translation.

Figure 3 depicts the compiler transformation necessary to synchronize the virtual addresses of shared pointers between CPU and GPU. Given the base addresses of CPU and GPU for the shared region as *cpu_base* and *gpu_base* respectively, a pointer *ptr_p* in the CPU virtual address space has a corresponding GPU virtual address *gpu_ptr_p* where $gpu\_ptr\_p = gpu\_base + (ptr\_p - cpu\_base)$. This address translation can be optimized by using the runtime constant $svm\_const = gpu\_base - cpu\_base$ that is computed only once. Then, before dereferencing *ptr_p* on the GPU, it can be translated to *gpu_ptr_p* by simply adding the runtime constant *svm_const*. Section 4.1 describes how we further optimize away part of this translation.

The right hand side of Figure 1 presents the compiler generated OpenCL code for the `operator()` function using the pointer transformation described in this section. The OpenCL kernel `operator_1` takes additional arguments for *gpu_base*, *cpu_base*, and the pointer *cpu_ptr* to the `Body` object (which is same as *b* in the source program). The shared pointers, *cpu_ptr* and $gpu\_ptr\_p[i].next$ are translated from the CPU address space to the GPU address space using the `AS_GPU_PTR` macro.

Our pointer translation technique can be generalized to scenarios where CPU and GPU use different encoding schemes and lengths. For example, if CPU memory is addressed using 64-bits and GPU memory uses 32-bits, we can apply the same pointer arithmetic as long as the shared region does not exceed 4GB.

3.2 Virtual functions

One of the most widely used dynamic features of C++ is its virtual function support. Although there are a variety of different ways to implement virtual functions, the *vtable* (virtual table) approach is common in modern C++ compilers. In this approach, a compiler creates a separate vtable for each class and, when creating an instance of that class (an object), adds to that object a pointer to the class's vtable. A call to a virtual function is then handled by dereferencing the underlying runtime object's vtable pointer, locating the corresponding virtual function entry and finally dereferencing that pointer to call the function. To implement virtual functions on the GPU, vtables need to be allocated in the shared region and, more importantly, function pointers are required on the GPU. Current integrated GPU hardware designs are not yet capable of supporting function pointers, so we use a compiler-based solution.

To support virtual functions on the GPU, the Concord compiler implements three key operations: a) move necessary vtables and runtime-type information to the shared region; b) share the global symbols of relevant virtual functions between the CPU and GPU using shared memory; c) translate a virtual function call into an inline sequence of tests of the call target against the possible target function pointer values for that call. The compiler implements global symbol sharing between CPU and GPU by allocating a new structure in the shared memory region that encapsulates all global symbols needed for the virtual function calls executed by a GPU function. It also determines the set of call targets for a given virtual function using class hierarchy analysis and alias analysis.

3.3 Reduction

When using `parallel_reduce_hetero`, the `Body` object's `join` method contains reduction code that combines two `Body` objects. We modified our compiler and runtime to perform hierarchical reduction of the body objects on the GPU using *local memory*, the high-speed on-GPU memory that is shared among all work-items of a work-group in OpenCL.

Our compiler generates OpenCL code for the `join` method similar to the code generation technique described in Figure 1. We generate additional wrapper OpenCL code that makes multiple copies of the shared `Body` object in each thread's private memory, invokes the `operator()` function to compute the thread's value that participates in reduction, moves the private objects to local memory, and finally, iteratively performs reduction using local memory until a single value is left. The local memory copies hold intermediate reduction results. The final reduced value is copied back to the original shared `Body` object. The original sequential `join` function pointer is also passed to the runtime to perform sequential reduction if local memory is insufficient or if the GPU is busy.

3.4 Code generation

The Concord compiler translates `parallel_for_hetero` and `parallel_reduce_hetero` to the runtime API functions `offload` and `offload_reduce` respectively. These runtime functions take additional compiler-generated arguments: (1) a `gpu_program_t` structure for the entire program to hold the OpenCL code and its cached JIT-compiled GPU binary; (2) a `gpu_function_t` structure to cache per-function GPU binary code in order to reuse the JIT-compiled code. The `gpu_function_t` also carries the user specified device information per kernel as specified in the third argument of `parallel_for_hetero` and `parallel_reduce_hetero`.

[1]On Intel's 4^{th} Generation Core processor, all *physical* memory is shared between CPU and GPU.

4. COMPILER OPTIMIZATIONS

The large register files offered by GPU architectures must be utilized by the compiler to improve performance. Classical compiler optimizations such as sub-expression elimination, aggressive register promotion, and loop-unrolling must be applied in order to exploit the register files. Given that shared pointers incur some overhead during translation, register promotion should be applied aggressively to eliminate memory loads of the same location, in particular, across loop iterations. Since loop-unrolling eliminates the overhead of address calculation and control instructions, we perform unrolling and control the unroll-factor by restricting *max_live* to the available physical registers.

Currently, our compiler promotes stack-allocated objects and reduction-based private copies of the body objects to private memory. These objects are not shared across threads since each thread creates its own instance. We also use local memory for performing reductions. Although it may make sense to also use local memory for C++ applications that reuse data among several GPU threads, the irregularity in the applications (which is the focus of our paper) makes it harder to perform it automatically in the compiler. Additionally, a language-based approach to support local memory in C++ has the disadvantage that the same C++ function cannot execute seamlessly between the CPU and GPU which is one of the key objectives of Concord.

Apart from the above standard compiler optimization techniques, we devise two new optimizations in Concord. The first optimization reduces the S/W-based SVM implementation overheads and the second optimization reduces cache contentions among multiple cores of the GPU. These two compiler optimizations are described in details below.

4.1 Reduce SVM implementation overhead

The pointer arithmetic operations inserted as described in Section 3.1 must be minimized by the compiler whenever possible. Depending on how shared pointers are used on the GPU, it may be beneficial to retain the CPU virtual address representation for a shared pointer instead of eagerly translating it to GPU address space. For example, if the GPU code loads a shared pointer and stores it into a memory location without dereferencing it, then it is better to never convert the CPU virtual address. Similarly, there are some situations where it is better to eagerly translate CPU to GPU addresses, and others where lazy translation is better. For example, consider the code sample shown in Figure 4.

```
1  int **a = data->a, **b = data->b;
2  for(int i=0; i<N; i++)
3     b[i] = a[i];
4  // a is not used on GPU after this
```

Figure 4: Example illustrating compiler transformation of shared pointers on GPU: *lazy* vs. *eager*

In this code fragment, pointer $a[i]$ is loaded from memory and written into $b[i]$ at each iteration of the loop. With eager translation (*i.e.,* convert to GPU virtual memory representation as soon as the pointer is loaded), we need pointer arithmetic operations to translate the array addresses a and b only immediately after their definitions, which are outside the `for`-loop. Using lazy translation (*i.e.,* keep the CPU virtual memory representation as is and translate to GPU representation just before dereferencing it), we must add pointer arithmetic to translate a and b from the CPU to the GPU representation on every loop iteration. The eager approach is clearly beneficial in this case.

On the other hand, eagerly converting the address of an array element $a[i]$ to a GPU virtual address results in wasted work because $a[i]$ is never dereferenced on the GPU. It would convert all $a[i]$ pointers to GPU addresses only to immediately convert them back to CPU addresses in order to store them in array b. The lazy approach is preferable in this case.

Both eager and lazy approaches have their advantages and disadvantages and can perform better or worse depending on the code patterns in a program. We devise a strategy where we keep both the CPU representation and GPU representation for every pointer. The GPU representation is obtained by converting the pointer eagerly when it is loaded from memory. If at a later use the pointer is stored into a memory location (as $a[i]$ in Figure 4), we replace the use by the CPU representation. Otherwise, we use GPU representation. If a pointer is never dereferenced on the GPU, a standard dead code elimination pass eliminates the redundant conversion to GPU address space. We optimize the placement of GPU pointer conversion operations using standard live-range shrinking techniques used in optimal code motion [23].

4.2 Reduce GPU cache-line contention

GPUs include a large number of hardware threads that execute concurrently and may access data from the global memory. GPU hardware typically coalesces global memory accesses from a workgroup to hide the latency of global memory access. Additionally, these accesses may be cached in the GPU cache hierarchy. The integrated GPUs use an unified L3 cache for all GPU cores to cache global memory accesses. This cache is not banked and thus suffers from contention among multiple GPU cores trying to access the same data in a cache line at the same time. We devise a compiler-based transformation in which we minimize the number of simultaneous accesses to the same cache line from multiple GPU cores.

Figure 5 depicts our loop transformation to reduce GPU cache-line contention. The `operator()` function on the left hand side has a loop that iterates over same array elements of a across multiple iterations of i. If iterations i and $i+1$ are executed on two separate GPU cores, then they will access the same array elements of a in the same order. This will result in increasing cache line contention. If the number of read and write ports to a cache line is not the same as the number of GPU cores (which is always the case), some cores will have to access the cache-line in a serialized fashion. On the other hand, the `operator()` function shown on the right hand side of Figure 5 does not suffer from this problem. Note that, W represents the number of GPU cores. The key idea is to ensure that the j loop is accessed in a different order for each GPU core. We apply this transformation to innermost loops.

5. EXPERIMENTAL EVALUATION

This section evaluates the Concord system using a set of irregular data-parallel C++ programs. We present comprehensive performance and energy measurements for these workloads using the GPU as well as CPU-only execution. We also show the impact of our compiler optimizations on

```
1  class LoopBody {
2    float *a;
3   public:
4    LoopBody(float *arr) : a(arr) {}
5    void operator()(int i) { // executed in parallel
6       ...
7       for (j=0; j<N; j++) { //innermost loop
8         ... = a[j];
9       }
10   }
11 };
```

$\Longrightarrow$

```
1  class LoopBody {
2    float *a;
3   public:
4    LoopBody(float *arr) : a(arr) {}
5    void operator()(int i) { // executed in parallel
6       ...
7       int start = i / W; // W: no. of GPU cores
8       for (j=0; j<N; j++) { //innermost loops
9         j_tmp = (j + start) % N;
10        ... = a[j_tmp];
11      }
12   }
13 };
```

Figure 5: Demonstration of loop transformation to reduce cache-line contention among GPU cores

performance and energy. Finally, we study the overhead of our software-based SVM implementation.

5.1 Experimental setup

We evaluated our Concord framework on two systems with integrated Intel 4^{th} generation Core processors running the Windows 7 64-bit operating system: (1) a 1.7GHz Dual-Core i7-4650U Ultrabook with 4GB memory, and (2) a 3.4GHz Quad-Core i7-4770 desktop with 8GB memory. The processor in (2) targets high-performance desktops and servers whereas the processor in (1) is a mobile processor that targets laptops and other mobile devices. While the desktop processor has a higher TDP (thermal design power) budget of 84W, the Ultrabook operates at a low TDP budget of 15W. Energy efficiency is particularly important for mobile systems such as the Ultrabook as it increases battery life. The integrated GPUs on the two systems each have 7 hardware threads, each of which is 16-wide SIMD. The desktop GPU is an Intel HD Graphics 4600 with 20 execution units (EUs) and runs at a turbo-mode controlled frequency from 350MHz to 1.25GHz. On the other hand, the Ultrabook GPU is an Intel HD Graphics 5000 with 40 EUs and runs at a turbo-mode controlled frequency from 200MHz to 1.1GHz.

Our evaluation used several irregular, data-parallel C++ workloads that use pointers extensively. Most of these were ported from existing TBB or multi-core C++ programs. Some were taken from the Galois system [2], the Rodinia benchmark suite [9], and OpenCV [7], while others we wrote ourselves. The origins and static characteristics of the workloads are presented in Table 1. We compiled all workloads using CLANG and LLVM version 3.3 with Concord extensions and using optimization level -O2.

We summarize the workloads below:

1. *BarnesHut:* This program uses the efficient Barnes-Hut algorithm for n-body simulation. It partitions the bodies into subregions using an *octree* so that forces from nearby bodies are computed exactly while forces from far-away particles are approximated. We target force calculations to the GPU. Since the octree is unbalanced and traversed recursively to compute the force on each body, the code is highly irregular.

2. *Breadth-first search (BFS):* This program does a breadth-first search in a graph that computes the distance of each node from a specified source node. It uses a compressed row representation, and so exhibits memory irregularity that depends on the input graph. Our results are for the Western USA road network.

3. *BTree:* This workload uses an n-ary search tree with records stored on leaves of the tree. Searching is targeted to the GPU. Since the search tree is unbalanced, the search process is irregular.

4. *ClothPhysics:* The applications models cloth soft-body using a graph consisting of distinct points (nodes) joined by springs (edges). As the cloth moves, new tension and torsion forces are computed for every node by traversing the neighboring nodes.

5. *Connected Component:* This program executes a topology-driven search in a connected-component graph. The search depends on the input graph and so is irregular.

6. *Facedetect:* This program detects faces using haar-like features that encode information about the faces. A cascade of classifiers is first trained and then applied to an input image. That cascade data structure is traversed during face detection process. The workload comes from OpenCV [7] computer vision library.

7. *Raytrace:* The key data structure used in raytracing algorithms is a *scene graph* consisting of objects and lights, each represented using a pointer vector. Our program uses a parallel version of the algorithm in [1]. During each pixel's color computation, scene graph components are intersected several times. Virtual function dispatch is used to intersect objects.

8. *Skip_list:* A skip list stores a sorted list of values using a hierarchy of linked lists, which enables efficient searches in $\mathcal{O}(log\ n)$ steps. While searching for values, this program traverses the intermediate linked-list structures that depend on the input data.

9. *Single source shortest path (SSSP):* This application uses the Bellman-Ford algorithm to compute the shortest path of all nodes from a fixed start node in a directed graph with weighted edges. It exhibits irregular memory access patterns that depend on the input graph.

Pointer-based data-parallel programs are traditionally difficult to port to GPUs without significant software engineering effort. However, since Concord supports pointer sharing using SVM and provides TBB-like APIs, we ported most benchmarks with little effort. Once ported, the *same* C++ code could run transparently on either the CPU or GPU. As an example of the effort involved, one of us ported the

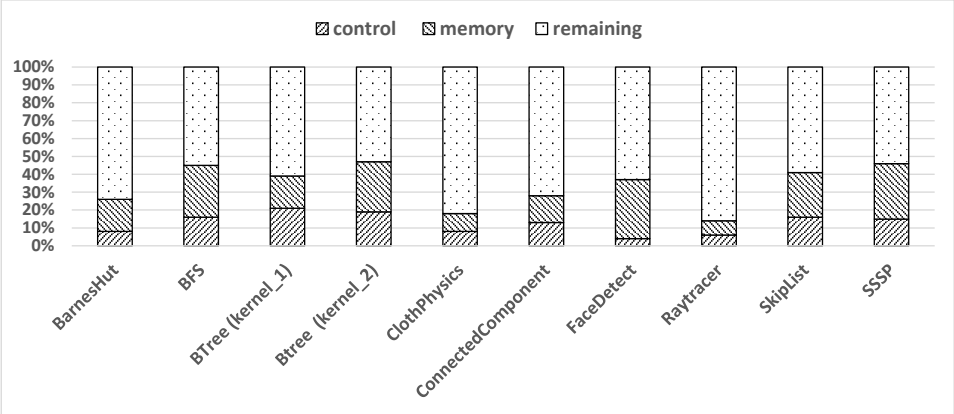

Figure 6: Percent of IR operations that are control-flow and memory related. A memory related operation is either a load or a store operation.

`ClothPhysics` application, which consists of 9234 lines of TBB code, to Concord in one day without any prior experience with it.

Irregular workloads tend to show large amount of control flow divergence or uncoalesced memory accesses. To understand the irregularities in our workloads, we collected static measurements of irregularity at the IR level. Figure 6 shows the percent of control flow and memory operations for each workload. In many cases, the sum of the control flow and memory operation percentages is quite high: more than 25%, which indicates that more than one in four IR instructions is either a control flow or memory instruction.

We report execution times and energy use for all benchmarks without any hand optimization after the port to Concord. We averaged the runtime performance of five runs. Our CPU execution times do not include compilation whereas our GPU execution times include a one-time compilation for each kernel. That is, multiple invocations of a kernel use the cached GPU binary as described in Section 3.4. We performed energy measurements by using an internal tool that measures package energy by sampling the machine-specific register MSR_PKG_ENERGY_STATUS.

Both energy improvement and speedup results relative to multi-core CPU execution are shown for the following cases:
1) `GPU` – GPU execution using Concord
2) `GPU+PTROPT` – GPU execution after applying the pointer transformation described in Section 4.1
3) `GPU+L3OPT` – GPU execution after applying the cache-line contention optimization described in Section 4.2
4) `GPU+ALL` – GPU execution after applying both pointer transformation and the cache-line contention optimizations

5.2 Performance and Energy Efficiency

5.2.1 Ultrabook

Figure 7 reports performance of our workloads under various configurations on the Ultrabook. With GPU execution using the `GPU+ALL` configuration, we found performance improvement ranged from 1.11× to 9.88× with an average improvement of 2.5× compared to multi-core CPU execution. It is not surprising to see that all workloads show performance improvement from offloading work to the GPU since the integrated 40 EU GPU on this system is more powerful than the dual-core CPU. `Raytracer` in particular, achieves the best performance improvement of 9.88× as it exhibits the least amount of irregularity compared to other workloads (as shown in Figure 6).

Figure 8 reports energy efficiency. We observe savings ranged from 0.93× to 6.03× with an average savings of 2.04× using GPU execution compared to multi-core CPU execution. All workloads except `FaceDetect` show energy savings from GPU execution. `Raytracer` has the highest energy savings of 6.04× which is primarily due to its high performance on the GPU.

5.2.2 Desktop

Figure 10 reports energy efficiency on the desktop system. With GPU execution using the `GPU+ALL` configuration, we found an average energy savings of 1.69× compared to multi-core CPU execution. All workloads except `FaceDetect` show energy savings from offloading work to the GPU. GPU execution of `BFS`, `Raytracer`, `SkipList`, and `BTree` yield especially significant energy savings–2.94×, 3.52×, 2.27×, and 2.43×, respectively–compared to multi-core CPU execution.

Interestingly, GPU execution results in significant energy savings even though it gives on average only 1% performance benefit (as shown in Figure 9) compared to multi-core CPU on the desktop system. The discrepancy between performance and energy efficiency on GPU vs. CPU is especially pronounced for `BarnesHut`, a tree traversal algorithm where the memory coalescing opportunity for two neighboring iterations of the `parallel_for_hetero` loop may depend on the input data. This workload is 47% slower on the GPU than the multi-core CPU and yet it is 48% more energy efficient.

On the desktop system, similar performance on the CPU and GPU for irregular workloads is not surprising since (1) the CPU cores have much higher main memory bandwidth than the integrated GPU cores, and (2) the CPU cores are equipped with highly accurate branch predictors that handle control flow divergence very well. Thus, even though there is large amount of parallelism on the GPU, GPU performance is hindered by application irregularity.

5.2.3 Discussion

`FaceDetect` is the only workload for which GPU execution does not reduce energy use on both systems. In `FaceDetect`,

Benchmarks	Origin	Input size	LoC	Device LoC	Data structure	Parallel construct
BarnesHut	In-house	1000000 bodies	828	105	tree	parallel_for_hetero
BFS	Galois [2]	W-USA (|V|=6262104, |E|=15248146)	866	19	graph	parallel_for_hetero
BTree	Rodinia [9]	command.txt	3111	84	tree	parallel_for_hetero
ClothPhysics	Intel [8]	50K nodes & 200K connections	9234	411	graph	parallel_reduce_hetero
ConnectedComponent	Galois [2]	W-USA (|V|=6262104, |E|=15248146)	473	36	graph	parallel_for_hetero
Facedetect	OpenCV [7]	3000x2171, Solvay-1927	3691	378	cascade	parallel_for_hetero
Raytracer	In-house (alg. in [1])	sphere=256, material=3, light=5	843	134	graph	parallel_for_hetero
Skip_list	In-house	50000000 keys	467	21	linked-list	parallel_for_hetero
SSSP	Galois [2]	W-USA (|V|=6262104, |E|=15248146)	1196	19	graph	parallel_for_hetero

Table 1: Concord C++ workloads and their characteristics

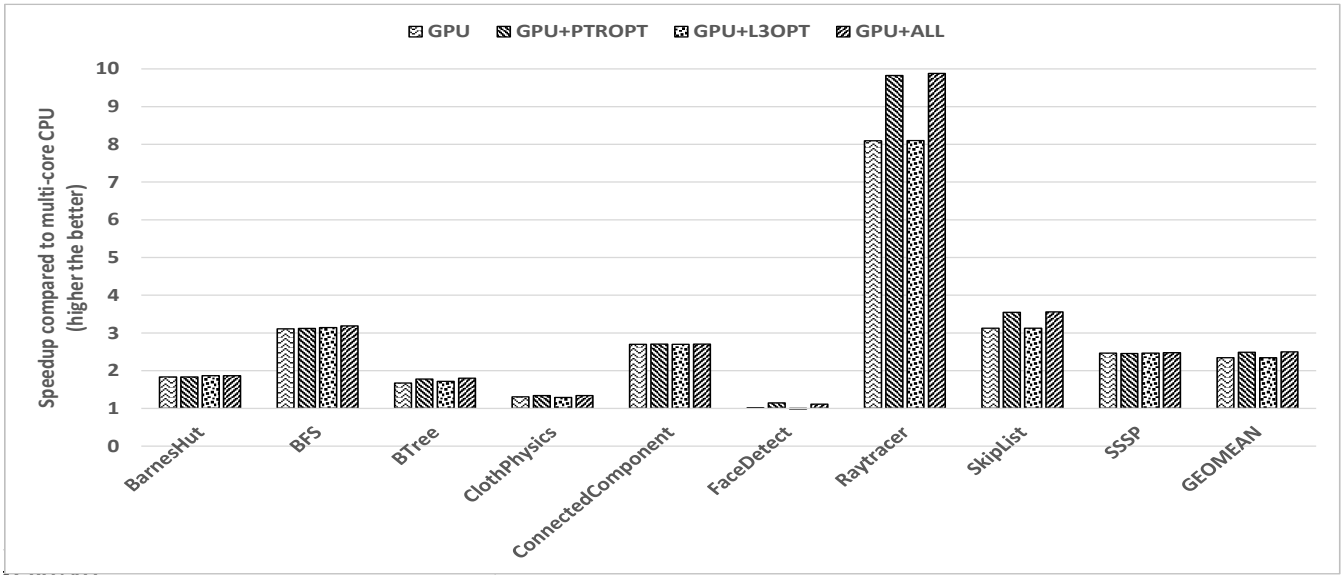

Figure 7: Runtime performance relative to multi-core CPU execution on the Ultrabook

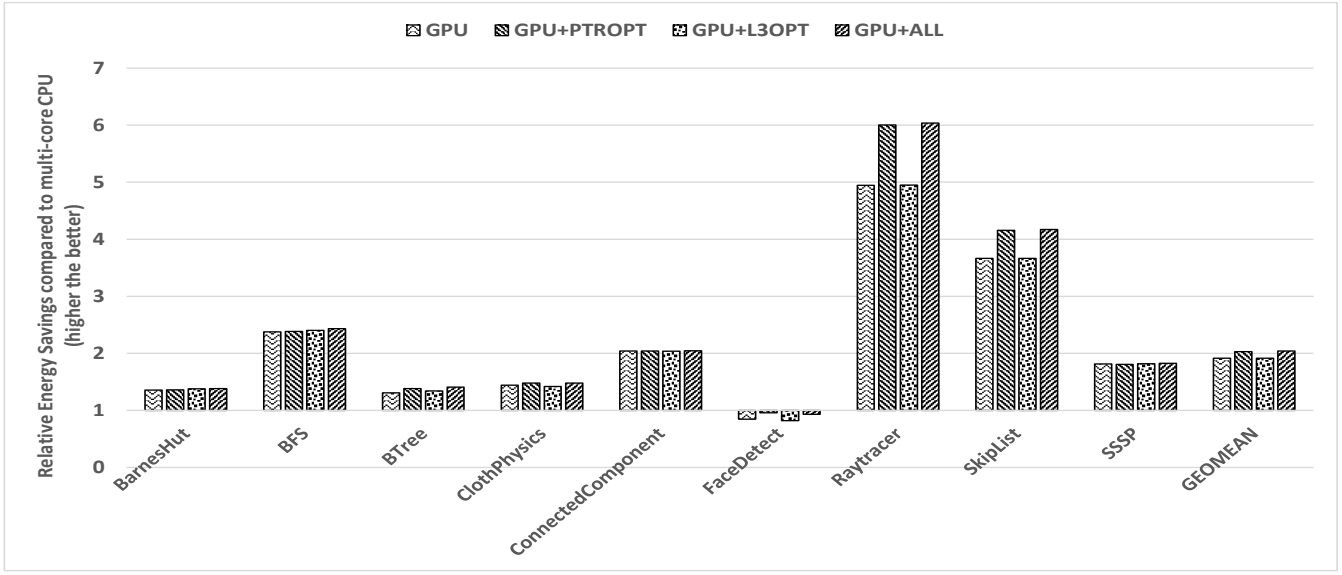

Figure 8: Energy efficiency compared to multi-core CPU execution on the Ultrabook

each `parallel_for_hetero` loop iteration moves though a series of 22 stages where, depending on the input image, each iteration may either continue to next stage or abort. The highly-dynamic behavior this benchmark exhibits is not well-suited for GPUs. In contrast, workloads like `Raytracer` perform well on the GPU since they are less irregular as shown in Figure 6.

5.3 Effect of Compiler Optimizations

We now discuss the impact of the two compiler optimizations described in this paper.

5.3.1 Ultrabook

The pointer arithmetic optimization described in Section 4.1 eliminates much of the overhead of pointer arithmetic and yields in an average performance improvement of 1.06×, as shown in Figure 7 for the `GPU+PTROPT` configuration compared to `GPU` execution on the Ultrabook. For `Raytracer`, `FaceDetect`, and `SkipList`, we observe significant performance improvements–1.21×, 1.13×, and 1.13× respectively–due to the elimination of pointer arithmetic operations in deeply nested loops. Our second optimization, described in Section 4.2, reduces cache-line contention among GPU execution units. We did not observe any obvious performance improvement on our workloads by applying this optimization alone (`GPU+L3OPT` configuration). However, when it is applied along with the pointer arithmetic optimization (`GPU+ALL` configuration), we observed an average performance improvement of 1.07× compared to `GPU` configuration. The average effect of the combined optimizations on energy consumption is 1.07× energy savings.

5.3.2 Desktop

The pointer arithmetic compiler optimization yields an average performance improvement of 1.09×, as shown in Figure 9 for the `GPU+PTROPT` configuration compared to `GPU` configuration. As for the Ultrabook, `Raytracer`, `FaceDetect`, and `SkipList`, show significant performance improvements on the desktop as well: 1.34×, 1.16×, and 1.13×, respectively. Applying both the optimizations (`GPU+ALL`), we observed an average performance improvement of 1.12× compared to `GPU`. Furthermore, they save 1.12× energy compared to `GPU` configuration.

5.3.3 Discussion

The higher performance for the combined effect of both the compiler optimizations can be attributed to first eliminating the pointer arithmetic and then improving the scheduling of load and store instructions. On the desktop system, for `FaceDetect`, the combined effect, shown by `GPU+ALL`, is about 1.064× of `GPU+PTROPT`.

5.4 Overhead of our SW-based SVM

To study the overhead of our SVM implementation we took one pointer-intensive Concord workload, `Raytracer`, and implemented an equivalent OpenCL 1.2 program. Since OpenCL 1.2 doesn't support pointer sharing between the CPU and GPU, the OpenCL Raytracer's host CPU program had to flatten the pointer-based scene graph data structure, convert its embedded vectors into linear arrays, and create OpenCL buffer objects in order to share that scene graph with the GPU. In addition, the Concord `Raytracer` code executing on the GPU had to be translated to OpenCL C and modified to traverse the flattened scene graph representation using integer offsets. We found negligible overhead for small images while, for even the largest image size, we observed only a 6% overhead.

6. RELATED WORK

6.1 CPU-GPU Programming Frameworks

CUDA [6], OpenCL [4], and C++ AMP [5] program GPUs in a subset of C or C++, but differ in target platforms and SVM and language support. CUDA is popular for programming NVIDIA GPUs. It supports most of C++ and implements SVM support on NVIDIA's Fermi-class discrete GPUs via a combination of hardware and software techniques. Concord implements SVM purely in software targeting integrated GPUs with low cost CPU-GPU communication. In addition, CUDA programmers must write separate host and device code, while Concord provides a single programming model for the CPU and GPU. OpenCL is implemented on a variety of CPUs, GPUs, and accelerators. While OpenCL 2.0 includes SVM [4], OpenCL 2.0 implementations have yet to be released. Microsoft's C++ AMP [5] is C++-based and runs on GPUs that support DirectX. Like Concord, it supports data-parallel execution of loops on GPUs but it doesn't support SVM, virtual functions, and data-parallel reductions on GPUs.

Several other efforts have aimed to simplify offloading work to GPUs. Grewe et al. [19] developed a compiler to automatically generate optimized OpenCL code from data-parallel OpenMP programs. It automatically determines whether to run OpenCL code on the GPU or to run OpenMP code on the multi-core host. Pragma-based languages include OpenACC, which defines a set of compiler directives to specify loops and regions of code in standard C, C++, and Fortran to be offloaded from a host CPU to an accelerator; Mint [30], which uses pragmas and targets stencil computations; and HiCUDA [21], which is directive-based and generates CUDA code from a C-like language. Also, Lee et al. [26, 25] generate CUDA code from OpenMP pragmas. Baskaran et al. [14] developed an automatic code transformation system that generates parallel CUDA code for regular programs from sequential C code. They used polyhedral compiler optimizations to leverage the memory hierarchy. CnC-Cuda [20, 29] is graph-based parallel coordination language for heterogeneous platforms. Cunningham et al. [17] generate CUDA code from the X10 language and Dubach et al. [18] augment the Lime [13] programming language to target GPUs. Catanzaro et al [15] provide nested data parallel abstractions for Python on NVIDIA GPUs. None of these systems support SVM or transparent pointer-sharing betweeen CPU and GPU. Hera-JVM [27] and Offload C++ [16] deal with offloading Java and C++ respectively to the Cell processor.

6.2 Optimizations for Irregular workloads on GPU

Improving performance of general-purpose applications running on CPU and/or GPUs is also widely studied. Program performance on GPUs is impacted by irregularities. Zhang et al. [32] developed a framework to tackle these dynamic irregularities through data reordering, job swapping and hybrid transformation. Dynamic index reordering was also mentioned in [19]. Ren et al. [28] exploited data-parallelism in non-numeric, non-graphics applications,

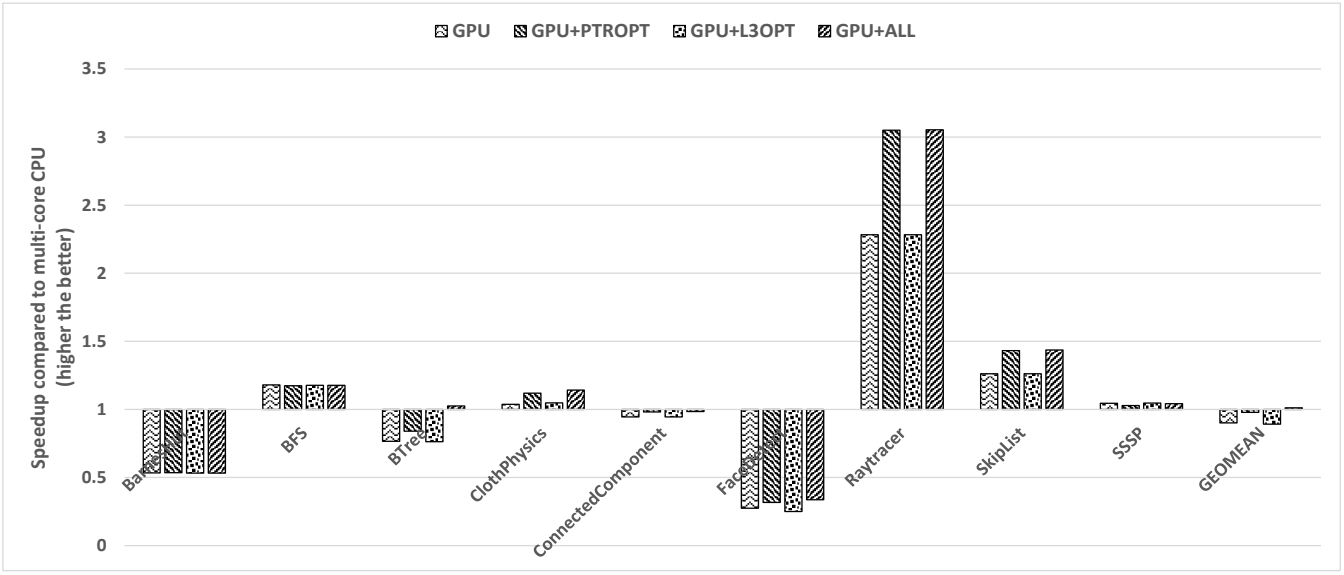

Figure 9: Runtime performance relative to multi-core CPU execution on the desktop system

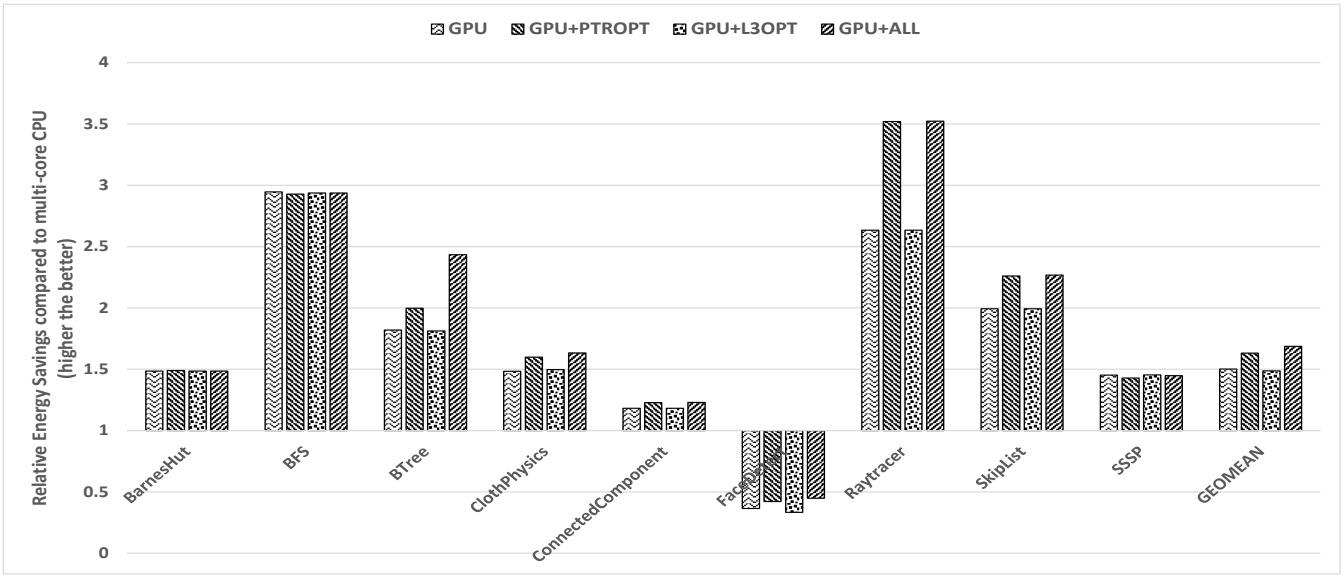

Figure 10: Energy efficiency compared to multi-core CPU execution on the desktop system

which transverse irregular data structures. They provided a virtual machine for traversal specification and traversal to SIMD units mapping. Together with some optimizations for better data access, their infrastructure achieved significant performance improvement for two irregular data structure traversing applications. Wu et al. [31] proposed a compiler to do unstructured-to-structured control flow transformation. Hong el al. [22] improved performance of some irregular workloads on GPU through virtual warp sizing. These optimizations are complimentary to Concord. Concord provides a simple and high-level programming framework for irregular application and in future, we will import good optimizations from the above works to keep improving its performance.

Memory hierarchy is also a research focus of heterogeneous computing. A thread-level parallelism (TLP)-aware cache management policy for CPU-GPU heterogeneous architectures was proposed in [24] to improve LLC sharing between CPU and GPU, which can be augmented to Concord to improve its performance.

7. CONCLUSIONS

A number of specialized languages have been developed for offloading work to GPUs, but their use has been restricted by their complexity and required architectural understanding. Furthermore, these languages are targeted at accelerating regular data-parallel applications operating on array-based data structures, not the kind of pointer-based applications typical in multi-core C++ programming that operate on irregular data structures such as trees and graphs. This paper describes the Concord C++ programming framework for processors with integrated GPUs. With its support for SVM and most C++ constructs, Concord is designed to allow object-oriented C++ data-parallel programs to take advantage of GPU execution. Its compiler optimizations reduce GPU cache contention and the cost of software-based SVM. Using nine realistic irregular C++ applications, we demonstrate that C++ applications using pointers and other object-oriented features can be automatically mapped to the GPU. Furthermore, we demonstrate that GPU execution can bring significant energy benefits to irregular applications even without sophisticated algorithm or data restructuring changes: our results show an average energy savings of 2.04× on an Ultrabook and 1.69× on a desktop over multi-core CPU execution. Much research has gone into improving the performance of regular data-parallel GPU applications. Our work on accelerating irregular C++ programs is complementary to this research, and could be combined with it for even better results.

8. ACKNOWLEDGMENTS

We would like to thank Stephan Herhut, Neal Glew, and Youfeng Wu for their valuable feedback on a preliminary draft of this paper that helped to improve its presentation. Finally, we would like to thank the anonymous reviewers for their comments and suggestions.

9. REFERENCES

[1] First-Rays. http://www.codermind.com/articles/Raytracer-in-C++-Part-I-First-rays.html.
[2] Galois. http://iss.ices.utexas.edu/?p=projects/galois.
[3] Intel Corporation. The Intel Thread Building Blocks. http://threading-buildingblocks.org/.
[4] Khronos OpenCL Working Group. The OpenCL Specification, http://www.khronos.org/opencl/.
[5] Microsoft Corporation. C++ Accelerated Massive Parallelism Specification.
[6] NVIDIA Corporation. The CUDA Specification, http://developer.nvidia-.com/object/cuda.html.
[7] OpenCV. http://sourceforge.net/projects/opencvlibrary/.
[8] Petme. http://software.intel.com/en-us/articles/multi-core-simulation-of-soft-body-characters-using-cloth/.
[9] Rodinia. http://lava.cs.virginia.edu/Rodinia/.
[10] Task Parallel Library (TPL). http://msdn.microsoft.com/en-us/library/dd460717.aspx.
[11] The Cilk Project. http://supertech.csail.mit.edu/cilk.
[12] The OpenACC^{TM} Application Programming Interface, www.openacc-standard.org/.
[13] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. OOPSLA'10.
[14] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. CC'10/ETAPS'10.
[15] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. PPoPP'11.
[16] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload - automating code migration to heterogeneous multicore systems. HiPEAC'10.
[17] D. Cunningham, R. Bordawekar, and V. Saraswat. GPU programming in a high level language: compiling X10 to CUDA. X10'11.
[18] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). PLDI'12.
[19] D. Grewe, Z. Wang, and M. F. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. CGO'13.
[20] M. Grossman, A. S. Sbirlea, Z. Budimlic, and V. Sarkar. CnC-CUDA: Declarative Programming for GPUs. LCPC'10.
[21] T. D. Han and T. S. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *TPDS'11*.
[22] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. PPoPP'11.
[23] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: theory and practice. *TOPLAS'94*.
[24] J. Lee and H. Kim. TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture. HPCA'12.
[25] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. SC'10.
[26] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. PPoPP'09.
[27] R. McIlroy and J. Sventek. Hera-JVM: a runtime system for heterogeneous multi-core architectures. OOPSLA'10.
[28] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. SIMD parallelization of applications that traverse irregular data structures. CGO'13.
[29] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. LCTES'12.
[30] D. Unat, X. Cai, and S. B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated C. ICS'11.
[31] H. Wu, G. Diamos, J. Wang, S. Li, and S. Yalamanchili. Characterization and Transformation of Unstructured Control Flow in bulk synchronous GPU Applications. JHPCA'12.
[32] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. ASPLOS'11.