Helium: A Transparent Inter-kernel Optimizer for OpenCL

Thibaut Lutz
University of Edinburgh
10 Crichton Street
Edinburgh, UK
thibaut.lutz@ed.ac.uk

Christian Fensch
Heriot-Watt University
Riccarton
Edinburgh, UK
c.fensch@hw.ac.uk

Murray Cole
University of Edinburgh
10 Crichton Street
Edinburgh, UK
mic@inf.ed.ac.uk

ABSTRACT

State of the art automatic optimization of OpenCL applications focuses on improving the performance of individual compute kernels. Programmers address opportunities for inter-kernel optimization in specific applications by ad-hoc hand tuning: manually fusing kernels together. However, the complexity of interactions between host and kernel code makes this approach weak or even unviable for applications involving more than a small number of kernel invocations or a highly dynamic control flow, leaving substantial potential opportunities unexplored. It also leads to an over complex, hard to maintain code base.

We present Helium, a transparent OpenCL overlay which discovers, manipulates and exploits opportunities for inter-and intra-kernel optimization. Helium is implemented as preloaded library and uses a *delay-optimize-replay* mechanism in which kernel calls are intercepted, collectively optimized, and then executed according to an improved execution plan. This allows us to benefit from composite optimizations, on large, dynamically complex applications, with no impact on the code base. Our results show that Helium obtains at least the same, and frequently even better performance, than carefully handtuned code. Helium outperforms hand-optimized code where the exact dynamic composition of compute kernel cannot be known statically. In these cases, we demonstrate speedups of up to 3x over unoptimized code and an average speedup of 1.4x over hand optimized code.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments. Optimization. Incremental compilers*

Keywords

GPGPU, OpenCL, profiling, inter-kernel optimization, JIT compilation, staging

1. INTRODUCTION

Compute accelerators have become a common component in all areas of computation: from mobile phones to supercomputers. Examples of accelerators include specialized processors (Intel Xeon Phi), FPGAs (Altera Cyclone V) or GPGPUs (Nvidia Tesla). Most accelerators adopt a compute kernel based software development process in which the developer splits the application into two parts. Compute intensive parts are packaged into compute kernels that execute on the accelerator. The remainder of the application executes on the host and is responsible for managing communication to the device and dispatching computation. CUDA[1] and OpenCL[7] are two widely used implementations of this approach. The former is a proprietary approach by Nvidia limited to Nvidia devices; the latter is an industry standard that is widely supported by more than 15 leading hardware companies.

Most current research has focused on improving the performance of individual compute kernels[15, 9, 16]. However, most real world applications consist of multiple, interdependent compute kernels. Like traditional function calls, this composition of kernels could be subjected to interprocedural optimizations (such as inlining or loop fusion). Unfortunately these optimizations are not easily applied in practice because of the separation of concerns between host and device programs and the lack of unified analysis. While the host has some knowledge about the inputs and outputs of a compute kernel, it has no information about how the data is manipulated and the resulting data dependencies. Similarly, due to the compute kernel based execution model, the accelerator is only aware of the operations that the currently executing kernel performs. Furthermore, in some cases it is not possible to determine the precise composition of kernels statically as it relies on runtime decisions.

Applying these optimizations manually is sometimes possible since programmers have a global view of the application; however this approach has several drawbacks. Each instance of code specialization creates additional functions by duplicating or fusing existing kernels, introducing redundancies. This results in a code base that is more difficult to maintain. Furthermore, for some of these optimizations it is difficult to decide statically if they are beneficial for a particular device or application, so the high development costs represent a significant risk.

In this paper, we present Helium, a transparent OpenCL overlay that overcomes these obstacles and allows us to perform inter-kernel optimizations of unmodified, binary OpenCL applications. Intercepting all OpenCL API library calls, our system is able to build a dynamic task and data dependency graph of the OpenCL application. In addition, we postpone the execution of all OpenCL API calls until the host application requests an output from the accelerator. At this point, we use the collected task and data flow dependency information to perform inter-kernel based optimizations.

To the best of our knowledge, we are the first to perform this kind of optimization for compute kernel based applications. Our results show that we can improve application performance up to 3x, and 1.87x on average.

The contributions of this paper can be summarized as follows. It is the first to:

- Present an approach that automatically constructs dataflow graphs in any OpenCL application by combining compiler analysis and runtime information.

- Develop a delay mechanism for OpenCL commands and replay them lazily, transparently to the application.

- Dynamically apply effective inter-kernel optimizations such as task reordering and parallelization or kernel fusion.

The remainder of this paper is structured as follows: Section 2 motivates this work, Section 3 provides an overview of Helium, which is then detailed in Section 4. Section 5 and 6 present our evaluation methodology and its output. Section 7 lists state of the art research in this field and relevant techniques developed to solve similar problems. Section 8 discusses the implications of this work and exposes future extensions; and Section 9 concludes.

2. MOTIVATION

Complex applications often define computation as a stream of data through a set of independent operators, creating a modular and maintainable code base. Composable operators can be optimized very efficiently to improve data locality, by merging stages producing and consuming data, or eliminating redundant data accesses across multiple stages of the computational pipeline. For most programming environments, developers are oblivious to these optimizations since they are performed by the compiler, allowing programmers to focus on application semantics rather than optimization. In languages like C or C++, static compiler analysis finds the dataflow paths in the application and interprocedural optimization improves the implicit or explicit data flow whenever possible.

Sadly these optimizations do not arise naturally in OpenCL (or any other compute kernel based approach): since the code is broken down into host and device code, there is no global analysis of the data flow between host and accelerators. The host program does not know the computation patterns and how the data is being consumed by the compute kernels, while the kernels do not contain information about the data flow or the compute sequences. This prevents automatic inter-kernel compiler optimizations in either the offline host compiler or the runtime device compiler.

However the optimization potential of inter-kernel transformation is very high since improving data locality is considered to be one of the most effective optimization on many architectures. For most GPUs for example, data caches are not persistent across kernel executions, making redundant memory accesses very expensive and a significant waste of resources when several kernels manipulate the same input data or temporary intermediate buffers.

Because of this, inter-kernel optimizations are often performed by hand. Analyzing an OpenCL application manually requires a constant switching between host and device code, as well as keeping track of the various OpenCL objects across the application. This is a tedious, difficult and error prone task. The main obstacles are the extensive changes required for optimizing the computation and the difficulty to follow data flow paths between host and device.

The example shown in Figure 1 represents an application with multi-kernel asynchronous execution and dynamic dataflow. Before computation is delegated to the device, a number of steps are

```
 1    // Compile device source code
 2 🟪 std::string c = R"(
 3 🟪  #define ID get_global_id(0)
 4 🟪  #define buf global int*
 5 🟪
 6 🟪  kernel void A(buf a, buf b)
 7 🟪  { b[ID] = 2 * a[ID]; }
 8 🟪
 9 🟪  kernel void B(buf t, buf u)
10 🟪  { u[ID] = t[ID] - 1; }
11 🟪
12 🟪  kernel void C(buf x, buf y, buf z, int n)
13 🟪  { y[ID+n] += x[ID-n] + z[ID-n]; } )";
14 🟪 Program p{ctx,{1{c.data(),c.size()}}};
15 🟪 p.build(devices);
16
17    // Memory allocation
18 🟧 Buffer b1{ctx,CL_MEM_READ_WRITE,size};
19 🟧 Buffer b2{ctx,CL_MEM_READ_WRITE,size};
20 🟧 Buffer b3{ctx,CL_MEM_READ_WRITE,size};
21
22    // Kernel creation and argument binding
23 🟨 Kernel A{p,"A"}, B{p,"B"}, C{p,"C"};
24 🟨 A.setArg(0,b1); A.setArg(1,b2);
25 🟨 B.setArg(0,b1); B.setArg(1,b3);
26 🟨 C.setArg(0,b2); C.setArg(1,b1); C.setArg(2,b3);
27
28    // 'n' is a value computed at runtime
29 🟨 int n = computeN(); C.setArg(3,n);
30
31    // Enqueue non-blocking operations
32 ◽ q.enqueueWriteBuffer(b1,CL_FALSE,0,size,data);
33 🟨 q.enqueueNDRangeKernel(A,{0},{g},{1});
34 🟨 q.enqueueNDRangeKernel(B,{0},{g},{1});
35    if( n > 1 ) { // dynamic computation
36 ◽   q.enqueueReadBuffer(b2,CL_FALSE,0,size,tmp);
37 🟨   q.enqueueNDRangeKernel(C,{n},{g-n},{1});
38      n = 0;
39 🟨   C.setArg(3,n); C.setArg(0,b3); A.setArg(0,b2);
40    }
41 🟨 q.enqueueNDRangeKernel(A,{0},{g},{1});
42 🟨 q.enqueueNDRangeKernel(C,{n},{g-n},{1});
43
44    // Blocking operation
45 ◽ q.enqueueReadBuffer(b1,CL_TRUE,0,size,result);
46 🟧 if( n > 1 ) { b2 = Buffer(); }
```

Figure 1: Example of application using the C++ OpenCL API.
🟪 The device program is compiled at runtime from OpenCL-C.
🟧 Memory buffers are allocated/released explicitly by the host.
🟨 Kernels are configured and invoked in separate operations.
◽ Communication functions copy data between host and device.

necessary to set up the device application. First, the device program is compiled from source (🟪) and data is allocated on the device (🟧). Kernels objects are then created and their arguments are bound to host variables (🟨). Finally the data is copied to the device (line 32), then processed by a series of kernels (lines 33, 34, 37, 41 and 42) and the final result is read back (line 45). All these steps must be cross-referenced to find the actual dataflow paths. More specifically, the dataflow analysis must keep track of:

- *kernel states*: unlike C functions, OpenCL kernels have persistent states for their arguments, which do not need to be specified for each invocation.

- *data access patterns*: for each kernel invocation, one must refer to the device code with the latest known state of the arguments to know which memory object is updated or consumed.

- *synchronization primitives*: most of the OpenCL actions execute asynchronously, only special functions or blocking operations guarantee coherence of the data in the host program.

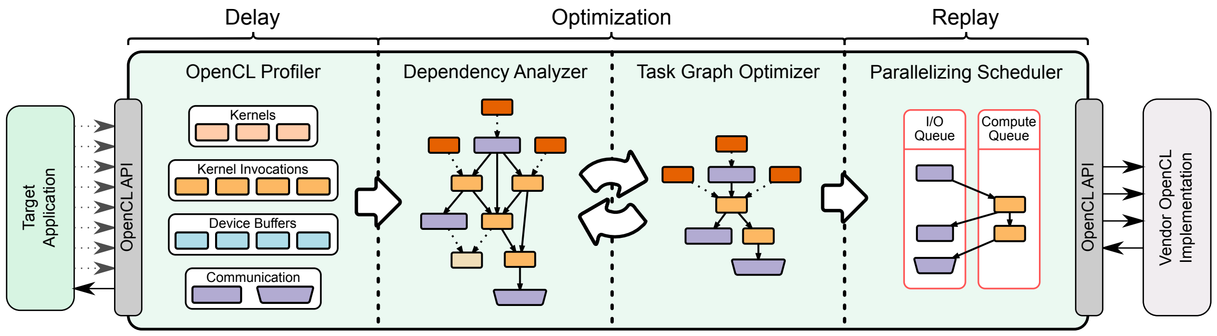


Figure 2: Overview of the Helium. The calls to OpenCL functions from a target application are intercepted to gather profile information. This information is then analyzed and combined to build a task graph, which is optimized before being executed by the vendor implementation.

- *conditional compute sequences*: kernels might be enqueued in branches or the state of the OpenCL objects might change according to the application's control flow. In the example, some computation depends on the runtime value of n.

Even if the dataflow can be traced statically, applying the optimizations to the code is equally challenging. Merging operators to improve data locality requires implementing new kernels, for which the business logic of the host program has to be adapted. This also creates many variants of the same code, which greatly decreases maintainability. Modifying the host code and introducing new kernels might introduce races in different parts of the application if the synchronization points or the kernel arguments are not correct. These problems are hard to debug or might go undetected; and because of this, inter-kernel optimizations are rarely applied or only on non-dynamic parts of the computation pipelines.

By contrast, Helium can dynamically trace a running application, find the minimal set of dependencies between OpenCL actions and optimize the task graph before it is transparently executed by the vendor implementation. New kernels are created on the fly from the initial set of kernels to improve data locality, and tasks are parallelized automatically to take advantage of the task parallelism provided by the OpenCL model.

3. HELIUM OVERVIEW

The Helium optimizer uses a *delay-optimize-replay* mechanism; by which all asynchronous OpenCL commands are postponed and executed lazily in order to exploit as much runtime information as possible and depict a broader execution plan spanning multiple kernel invocations. The optimizer then improves these execution plans before replaying them. This process takes place between the host program and the OpenCL vendor implementation, thus it can be deployed transparently over any existing application. The main steps executed by our system, represented in Figure 2, are:

1. *Profiling*: the execution context of each OpenCL command is saved at its call site, which corresponds to its parameters, and the call to the OpenCL function implementation is *delayed*.

2. *Analysis*: datapaths are computed by keeping track of the last command modifying each device memory object and connecting it to each subsequent action reading from it.

3. *Optimization*: when a command which has a side effect on the host program (either a blocking operation, a wait command or a barrier) is enqueued, the framework recursively builds an optimized task graph of all dependent commands.

4. *Scheduling*: the task graph is *replayed* in topological order using a parallelizing scheduler, exposing task parallelism. The host program is blocked until completion of all the actions required to restore consistency in the target application.

While delaying the OpenCL actions might cause some overheads since it prevents host and device from performing computation in parallel, most applications delegating computation to an accelerator actively wait for the result shortly after issuing a chain of commands in order to proceed further, hence it does not cause significant delays. Similarly, profiling overheads are minimal and are easily amortized by the resulting performance gain.

4. HELIUM IMPLEMENTATION

This section presents the implementation of the Helium OpenCL overlay and the optimizations it performes on the generated task graphs. Helium is packaged as a library, which can be preloaded when executing any OpenCL application and acts as a broker between the original program and the vendor OpenCL implementation, gathering runtime information and delaying the calls to the vendor implementation as much as possible.

Helium is divided in four components. We first describe in Section 4.1 what type of information is gathered by the *OpenCL Profiler*, then explain how it is used by the *Dependency Analyzer* in Section 4.2. The *Task Graph Optimizer*, detailed in Section 4.3, modifies the graph and the replay mechanism performed by the *Parallelizing Scheduler* is presented in Section 4.4. The code shown in Figure 1 will be used as a case study throughout the section to demonstrate the analysis and transformations. We consider an execution where the runtime value n is 2.

4.1 OpenCL Profiler

Before computing the data paths of an application, the analyzer must keep track of the OpenCL commands invoked and build an abstract representation for them in order to find their dependencies and the scope of the main OpenCL objects. To do this, Helium intercepts OpenCL function calls by overriding all the standard host functions and emitting information collected in the profiler. The main actions tracked by the profiler are:

Device Program Compilation. Most OpenCL applications compile the device program at runtime from OpenCL-C source code to improve portability. The profiler intercepts the source code through the OpenCL API calls. Helium's OpenCL compiler then analyzes this code to gather static information.

The usage of each kernel argument representing a pointer in the global memory space is traced across the kernel in order to find if the data is produced or consumed. If it used in a store instruction, it is annotated as *write*, and similarly if it is used in a load operation it is annotated as *read*. The result can be expressed as a map expression; for our example kernel A has two arguments: a and b. Analyzing this kernel finds a load from the argument a and a store from the argument b, which can be expressed as a map from a to b in kernel A; denoted $A : a \mapsto b$.

For each memory access, the compiler also builds a partially evaluated expression representing the offset in bytes from the base pointer used for the actual operation. For simplicity we present the linearized global position as a special *id* marker, which is used for evaluation and comparison of these expressions in the optimization stage. Both kernels A and B only access addresses with an offset *global_id(0)*, which is simplified to id. Kernel C uses both $id + n$ and $id - n$, where n is a kernel argument unknown statically. This information can be integrated with the previously described notation, and the output of the analysis for the code from Figure 1 can be expressed as:

$$A : a_{id} \mapsto b_{id} \qquad B : t_{id} \mapsto u_{id}$$
$$C : x_{id-n}, y_{id+n}, z_{id-n} \mapsto y_{id+n}$$

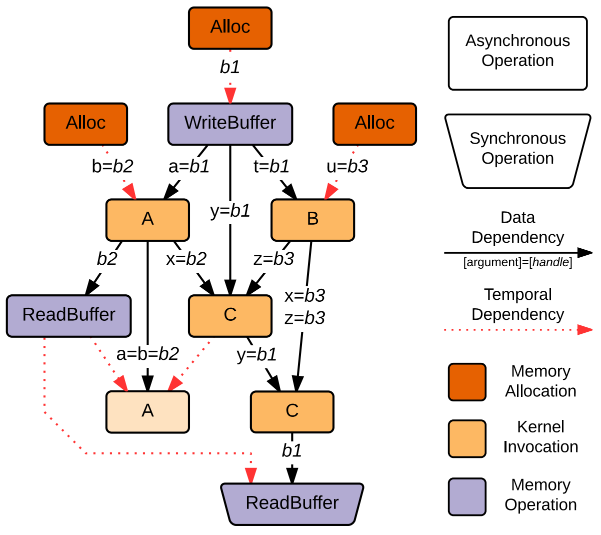


Figure 3: Task graph generated from analyzing the dataflow paths in the application from Figure 1 when n is 2. Each node is an OpenCL command and an edge represents a dependency.

Kernel Objects. Kernel objects in the OpenCL API represent a handle on a function executed on the device. They are created from a compiled OpenCL-C or a binary program using the name of a kernel function. Unlike native functions, the arguments of a kernel are persistent across calls and are independent from the call site. Each argument is set through a separate non type safe API where programmers set the raw binary content of each argument and its size. Hence, OpenCL kernels have a state which changes over the course of the application and often cannot be known statically.

Kernel Invocations. Kernel invocations are the equivalent of a function call on the device. They execute asynchronously a given kernel over an N-dimensional compute grid indicating the local and global sizes as well as the starting offset. The last state of the kernel arguments is used for the current invocation.

Task Dependencies. Kernel invocations, and other OpenCL actions like data copy, are issued in command queues which process requests either in-order or out-of-order depending on their properties. Each action in the queue can have explicit dependencies to other commands in the same or a different command queue.

Buffer Allocation and Deallocation. The OpenCL framework operates as a distributed memory model where the devices are passive. Device memory must be explicitly allocated by the host, which is also responsible for ensuring consistency of the buffers and their deallocation. Since allocation and deallocation are done through the OpenCL API, Helium can keep track of the lifetime of each individual memory object. This information can be used to improve the memory management: buffers can be allocated lazily just before being used, and freed immediately after their last use. In the example code, three buffers of identical size are allocated on the same context: $b1$ to $b3$.

Synchronization Operations. Most OpenCL commands execute asynchronously, and the host program is responsible for explicitly issuing synchronization to guarantee the coherency of the computation. A synchronous operation, which is achieved either by enqueuing a blocking command or using a synchronization primitive, blocks the host program execution until the asynchronous commands have completed.

When these commands are intercepted by the profiler, they are not forwarded to the vendor implementation straight away: this is the *delay* phase. Instead, each action generates an object representation within Helium. The objects encapsulate the action by copying their arguments and the *object handles* are replaced with virtual ones. Object handles represent pointers to an opaque type defined by a vendor implementation for representing allocated OpenCL objects. They are used to symbolize relationships between objects through the OpenCL API. For example, *clCreateBuffer* allocates memory within a context and returns a *cl_mem* handle, which can later be used to specify a memory operation or a kernel argument. However, a delayed buffer allocation does not have a valid handle in the vendor implementation, thus it must be replaced by a temporary virtual handle, which is returned to the target application and used transparently as a vendor provided one.

The output of the delay phase creates a set of commands objects, represented as the nodes in the Figure 3, without any connection. The dependencies between these actions are inferred by Helium's dependency analyzer.

4.2 Dependency Analyzer

The dependency analyzer uses the runtime information gathered from intercepting the OpenCL API function calls to build an abstract representation of the program and its execution flow. The result is a task graph of inter-dependent OpenCL commands, presented in Figure 3. The mechanisms used to build the dependency graph are described in the remainder of this section.

OpenCL Handle Tracking. Most of the relationships between objects are defined in the OpenCL API through the use of handles. However the handles do not directly provide dependencies between the actions; instead, they act as a common pool of objects used by different actions. Helium combines this information with semantic knowledge of each action to infer dependencies between them by linking each action to all OpenCL objects it manipulates.

Kernel invocations can be indirectly associated to memory objects through their arguments. When a kernel argument is set, its binary content is cross-referenced against the profiler's handle lookup table to determine whether it is an OpenCL object handle, a virtual handle or raw data. If it is a handle, the kernel is temporarily associated to the corresponding object, until the argument is overridden. When a kernel is invoked, its parameter list is copied to the invocation object, along with its relationships to other objects.

By tracking OpenCL handles, and in particular handles on allocated device memory, Helium can infer relations between actions by adding data and temporal dependencies.

Data Dependencies. A data dependency represents a producer-consumer dependency: an action creating or modifying data must be completed before the data is read again. Helium classifies each OpenCL action into producer or consumer categories. For example, allocating a buffer or writing data to the device *produces* input for the kernels, whereas releasing a buffer or reading data *consumes* it by acquiring its state.

Data dependency for kernels is more fine-grained, since combining the static device code analysis and the runtime value of each kernel argument for each invocation allows Helium to deduce if a particular buffer was produced or consumed. For example, at the first invocation of kernel A, the arguments were bound to $b1$ and $b2$ respectively. The static analysis for A derived by the profiler is $a_{id} \mapsto b_{id}$, which can be substituted with the runtime arguments: $b1 \mapsto b2$, hence this particular invocation consumed $b1$ and produced $b2$. It is then connected to the last actions producing the consumed buffer, here $b1$ was initialized by a memory write, so an edge is created between the two actions, as shown in Figure 3.

Our system can build the dataflow path in the application by connecting the last producer to all subsequent consumers for each device memory object.

Temporal Dependencies. After dataflow analysis, the resulting task graph contains only the minimal set of dependencies between actions. However it might still contain data races and ambiguities, which must be resolved by adding edges between conflicting nodes to enforce an ordering. Depending on whether the race was present in the original application or not, it can be solved in two ways:

- if the race was introduced by the delay replay mechanism, it can be solved using temporal dependencies. This occurs when the application expects an in-order queue to solve ambiguities, such as multiple unsequenced readers and writers. Helium uses the weak ordering of the original queue to insert a dependency edge from the earlier operations in the queue to the later ones, respecting the in-order semantics for conflicting nodes. In the example, the second invocation of A consumes and updates $b2$, which was last produced by the first invocation of A. However, the first invocation of C and a buffer read command also consume $b2$ and were enqueued before, so they must be completed before the second invocation of A is executed to prevent a data race, so two temporal edges are added to enforce this.

- if the race was present in the original application, it must have been resolved by the user. Since OpenCL supports task parallelism, race conditions naturally occurs in OpenCL when using multiple queues or out-of-order queues. To enforce an ordering, the OpenCL API provides events, which are handles optionally attached to each command. It is also possible to add a list of events as dependencies to an OpenCL action, which guarantees that all dependencies must be completed before the action starts. Our system analyzes these events and uses them to solve conflicts in the task graph, similarly to the OpenCL implementations. Note that all non-necessary event based synchronizations are discarded automatically: if there are user-specified dependencies between actions which are not conflicting in the task graph, they can be ignored as they cannot have visible side effects on the application and might slow down the execution.

Restoring Consistency. Finally, some additional edges need to be added to force the execution of asynchronous operations having a side effect on the host program. Executing only the ancestors of a blocking action is sufficient to restore consistency for this action. However the synchronization itself might trigger side effects for unconnected nodes, because it restores consistency between host and device. For example, enqueuing multiple asynchronous reads followed by a blocking read in an in-order queue guarantees that all pending reads will be completed when the blocking call finishes.

Helium solves this by tracking all operations having a side effect visible from the host and flushing them before a blocking call. In the example, the blocking read uses the buffer $b1$, for which the latest value can be computed independently of the asynchronous read in line 36. However, because the host program assumes consistency after synchronization, the read must be completed before the blocking call completes, so a temporal dependency is inserted between the asynchronous read and the blocking operation.

The resulting task graph, shown in Figure 3, is very specific to an execution where the value n is 2. The trace for other values might look completely different since some actions are executed conditionally. This highlights the difficulty of performing this analysis by hand, where all possible paths must be considered at once.

The resulting graph contains only a minimal set of dependencies and can be optimized before being replayed.

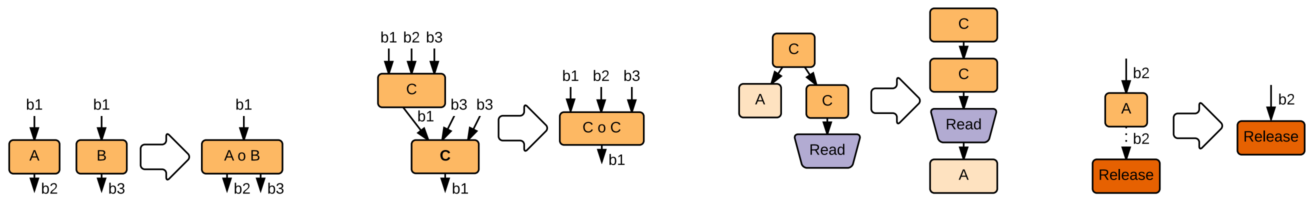
4.3 Task Graph Optimizer

Once the task graph has been built by combining the compiler analysis and the runtime information, it is passed to an optimizer before being replayed.

The role of the task graph optimizer is twofold: it aims to bridge the optimization gap between kernels which have been compiled in isolation, and maximize task parallelism. We classify the task graph optimizations in three categories: device code optimizations, which generate new kernels dynamically, host flow optimizations, which re-arrange tasks in the command queues in a more efficient way, and dynamic optimizations, which use profiling information to inject host runtime values to specialize device code. They will be described in that order in the remainder of the section.

Horizontal Fusion. When several nodes are at the same depth in the task graph, or more generally when there is no path between two nodes, these nodes are *data independent*. The absence of a path indicates that their relative ordering does not matter, or they can even be executed at the same time. If they are both compute nodes, their source code can be fused to improve data locality between the kernels. To find candidates for horizontal fusion, Helium groups nodes for which the input and output sets are disjoint. In the task graph presented in Figure 3, kernels *A* and *B* are data independent since their input and output set do not overlap:

$$(\{b1\} \cup \{b1\}) \cap (\{b2\} \cup \{b3\}) = \emptyset$$



(a) Horizontal Fusion (b) Vertical Fusion (c) Task Reordering (d) Task Elimination

Figure 4: Overview of the task graph optimization. Transformations (a) and (b) are device code optimization where specialized kernels are created on the fly. Optimizations (c) and (d) are host program modifications. Helium applies these transformations in the *optimize* phase.

This makes them a potential candidate for fusion. In this case, the optimizer will generate a fused kernel to improve data reuse: both *A* and *B* are reading from the same buffer *b1* at the same address *id*. The resulting kernel has one fewer parameter, as shown in Figure 4a. While there might be other occasions where horizontal fusion is beneficial without data reuse, such as amortizing the cost of spawning a new kernel or optimizing register occupancy, our system only generates fusion when there are common read instructions to amortize the compilation cost.

Vertical Fusion. A path between two nodes indicates a data dependency: the first node produces data, which is then consumed by the second, indicating a temporal relationship between the nodes. The OpenCL model provides two ways to avoid data races for temporal dependencies: either relying on the memory model or using memory fences. The memory model only guarantees sequential consistency in the global space within a single thread[1], so if there exists a mapping between the kernels such that data generated in each producer thread is read by at most one thread in the consumer, then the operation can be performed safely using the same thread for producer and consumer. If this mapping does not exist, a barrier is required to avoid data race, and since the OpenCL model does not provide a global memory fence or a global synchronization, the operations must be performed in two distinct kernels.

In the example, the first invocation of C is consumed by another instance of C, which creates a producer-consumer dependency in the task graph. In this case the conflicting buffer is $b1$, which is both read from and written to at address $id + n$. The variable n is a kernel argument, for which the value is known at each instance using the profiling information. The runtime can deduce that $b1$ is produced at $id + 2$ in the first instance and consumed at id in the other. Thus, merging the kernels directly would introduce spatial dependencies across kernel instances, which is not valid.

Using a technique similar to loop alignment and loop fusion, the optimizer determines that the alignment threshold of the fusion-preventing kernel is equal to 2. Therefore, the first invocation can be aligned with the second by adding the negation of the alignment threshold to the id function and adjusting the ranges. The first invocation of C now uses $id - 2$ between $[4, size)$ instead of id between $[2, size - 2)$, so the write operation now updates $id + 2 - 2$, which does not conflict with the following invocation of C. Hence, the spatial dependency has been eliminated, allowing the kernels to be fused. The fused kernel, shown in Figure 4b, has three input parameters instead of six, and a single store instead of two.

Because implementing this transformation by hand requires a modification of all memory accesses in a kernel from the device code, and an adjustment of the ranges each time the kernel is enqueued in the host code, it triggers cumbersome changes. Helium automatically adjusts the ranges dynamically and generates new device code where each fused kernel uses an adjusted index function with a range guard to ensure the correctness of the transformation.

[1]OpenCL 2.0 has a more complete memory model, but it is not yet supported by all vendors.

Task Reordering and Parallelization. Like horizontal fusion, task reordering is applicable between any two nodes which are not connected by data dependency. OpenCL supports Parallel task execution, provided that users carefully define the dependencies between the nodes to avoid data races. Since the task graph itself contains the minimal set of data dependency required, Helium automatically switches in-order command queues to out of order queues, and the edges in the task graph are converted to event handles, used to define dependencies. While this does not guarantee task parallelism, it is exposed through the OpenCL API. In our example, the asynchronous read operation is independent from the invocations of C, they will be enqueued without dependencies and may execute in parallel if the vendor implementation support it.

In order to reduce blocking delays, OpenCL commands are delayed as much as possible. Not all actions are always necessary to restore consistency in the host program; some can be delayed even after synchronization. The optimizer is able to delay actions across synchronization points whenever is it safe to do so, which concerns three types of actions:

- *computation nodes*: until one of the outputs is re-used in a chain of events having a side effect on the host. This increases the optimization potential of the next task graph following the synchronization operation. In the example the second invocation of the kernel A is not needed to evaluate the blocking operation, hence it is not replayed at this point.
- *write operations*: transferring data on the device is unnecessary until the data is consumed by a kernel. However host data might be discarded after synchronization, so the optimizer has to create a copy of at the synchronization point in order to replay it safely later. Asynchronous read operations cannot be delayed further since the host program assumes the transfer terminated after a blocking operation.
- *memory allocation/deallocation*: memory is a very limited resource on some devices, hence having an efficient memory management is often crucial. Memory allocation is delayed until the buffer is first used. Releasing memory is performed as soon as the last action using the buffer finishes, if the host released the object before the synchronization point.

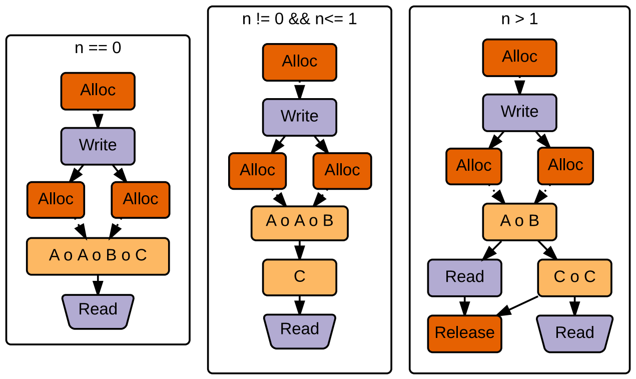


Figure 5: Optimized task graphs depending on the runtime value n. Implementing all the variations by hand would require a complex control flow in the host optimization and duplications in the device code.

Dead Task Elimination. Tasks having no side effect on the host program are often mistakenly introduced in complex applications as artifacts of the development process, especially after manually optimizing the code. Compute tasks can be considered unnecessary if their output is never read back in the host application or if is entirely overwritten by a subsequent computation before the data is ever used. In the example, the buffer $b2$ is used in a computation by the second invocation of Kernel A, but the buffer is released before the host reads back the result, meaning the computation had no side effect visible from the host and is not necessary. As a consequence, the second invocation of A will never be replayed. This optimization will occur automatically as a natural consequence of lazy instantiation of the actions. Task reordering was also necessary to delay the invocation of A to the next synchronization point.

Code Specialization. Creating specialized versions of the code by injecting runtime information into the device code is sometimes necessary. For example, when only one output of a kernel has no side effect, the compiler can eliminate the dead stores. However the remainder of the kernel has to be executed. In this case, the Helium compiler only removes the unnecessary store instructions and a dead code elimination pass will later simplify the code.

Code specialization is also used to propagate scalar values used in kernel arguments. This creates a highly specialized version of the code, which needs to be balanced with the compilation overhead: a scalar value that changes between kernel invocations, requires generating a new version of the kernel. Helium's default behavior is to specialize scalar values only if they control loops or large branches, since these are most likely to yield better performance.

Combining these optimizations incrementally generates a highly runtime specific task graph which takes into account the actual dataflow path. Figure 5 shows the three possible optimization sets depending on the value of n in the example application. If implemented by hand, the application would have required four additional specialized kernels: AoB, CoC, $AoAoC$ and $AoAoBoC$, with high code redundancy. This would have also considerably increased the complexity of the host application since each dataflow path must be implemented in different control flow path, which may extend beyond this code fragment to the rest of the application.

Once the task graph has been optimized, it must be scheduled on the device via the OpenCL vendor implementation.

4.4 Parallelizing Scheduler

The task graph scheduling corresponds to the *replay* phase. This is triggered by a synchronization operation from the host program. Helium must restore the consistency of the program in such a way that all actions having a side effect on the blocking operation must be replayed.

Command Queue Manager. Our system manages a set of out-of-order command queues to dispatch computation. Helium creates three separate queues per device for computation, read and write operations in order to maximize the chances of concurrency.

JIT Compilation. For the fused node, the newly generated kernels are compiled to using Helium's embedded compiler. The resulting program is loaded back into the OpenCL runtime as a native OpenCL binary program. As an orthogonal optimization, the program is also saved in an offline compilation cache along with the requirements of the transformation in order to accelerate future uses of the same fused set. In our example, the fused AoB kernel has no additional requirements and can be used any time the chain A followed by B is found. However, kernel CoC has been specialized in this particular case so the additional condition $n = 2$ for the first kernel and $n = 0$ for the second are both required since they are the values used for range alignment.

Replay Actions. These are achieved by recursively traversing the dependencies of the task graph from the blocking operation. Each node is enqueued only once in topological order. An event handle is attached to each action as they get enqueued, and the dependencies are translated to an array of event handles from the enqueued actions. This exploits task parallelism as expressible by the OpenCL framework: since the queue is out of order and the dependencies between actions are minimal, independent actions might execute concurrently if the vendor implementation supports it.

Kernel arguments must be rolled back to the state they were at the point of invocation, and restored immediately after the invocation has been issued to maintain consistency. Similarly, nodes resulting from a fusion have to be set up with the runtime values of the fused kernels. The argument of each kernel is copied and their dependencies are transferred to the fused node.

Profiler Update. Finally, the replayed commands transforming virtual handles into vendor specific ones are sent back to the profiler, and the handles are propagated through the task graph. Helium must seamlessly translate the virtual handles created from the delay phase into vendor specific handles to ensure correctness. Specifically, our system must translate every *cl_event* and *cl_mem* objects from a Helium assigned identifier to a vendor specific object. This also affects kernel arguments, for which the binary content must be updated if they contain an object handle.

The result of the replay phase is transparent to the target application. Figure 6 represents an equivalent OpenCL program to Figure 1 as seen from the vendor implementation point of view. The executed code contains fewer OpenCL API calls overall since the code has been optimized. The device code has been entirely rewritten in this case since the original kernels are not needed for the execution. Coding these optimizations by hand would require explicitly implementing all the possible control flow in separate branches along with the exact preconditions necessary for each optimized flow. This would lead to a high amount of code redundancy.

```
std::string c = R"(
 #define ID get_global_id(0)
 #define buf global int*
 kernel void AoB(buf at, buf b, buf u)
 { b[ID] = 2 * at[ID];
   u[id] = at[ID] - 1;}
 kernel void CoC(buf x, buf y, buf z)
 { if(ID >= 4)
     y[ID] += x[ID-4] + z[ID-4];
   y[ID] += 2 * z[i]; } )";
Program p{ctx,{1{c.data(),c.size()}}};
p.build(devices);

cl::Event e1, e2, e3, e4, e5;
Kernel AoB{p,"AoB"}, CoC{p,"CoC"};
Buffer b1{ctx,CL_MEM_READ_WRITE,size};
ioq.enqueueWriteBuffer(b1,CL_FALSE,0,size,data,
                                   nullptr, &e1);

Buffer b2{ctx,CL_MEM_READ_WRITE,size};
Buffer b3{ctx,CL_MEM_READ_WRITE,size};
AoB.setArg(0,b1); AoB.setArg(1,b2);
AoB.setArg(1,b3); CoB.setArg(0,b2);
CoC.setArg(1,b1); CoC.setArg(2,b3);
q.enqueueNDRangeKernel(AoB,{0},{g},{1},
                                   {e1}, &e2);
ioq.enqueueReadBuffer(b2,CL_FALSE,0,size,tmp,
                                 {e2}, &e3);
q.enqueueNDRangeKernel(CoC,{0},{g},{1},
                                   {e2}, &e4);
b2 = Buffer();
ioq.enqueueReadBuffer(b1,CL_TRUE,0,size,result,
                                 {e3, e4}, nullptr);
```

Figure 6: OpenCL code equivalent to the optimized application. The device code has been automatically re-written entirely taking into account dataflow aware optimizations.

5. EXPERIMENTAL SETUP

We evaluate Helium on a collection of applications, which are present in both raw (baseline) and hand-optimized forms. This enables us to evaluate the benefits Helium brings directly *against* hand-optimization and the benefits it brings when applied *after* hand-optimization. For complex and dynamic applications, opportunities for hand-optimization are quite restricted and the latter comparison demonstrates that they may even be counter-productive when a powerful, automated system such as Helium is available. We use the following four benchmarks:

- *CCO* is a simplified version of the copy-compute overlap benchmark from the Nvidia benchmark suite, demonstrating parallel computation and communication. The computation flow is the same for a given number of iterations: write two input buffers to the device, enqueue a kernel which uses both inputs and generates an output buffer, which it read at each iteration. The optimized implementation fragments the computation in by splitting the input and output buffers in half. The commands are then pushed in two in-order queues in a very specific order such that computation and communication may overlap across iterations. The baseline implementation is a simplified version of the code where all actions are pushed in a single in-order queue.

- *Sobel*: the Sobel filter is a discrete differentiation operator commonly used for image processing applications like edge detection. In its general form, two gradient convolution operators are applied to the input, generating two temporary values, which are combined by a third operation. In this case the code can be hand optimized but in more complex applications these operators are created by composition of simple filters, making the number of combinations impossible to optimize by hand.

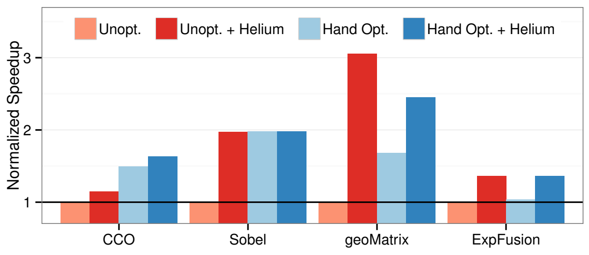


Figure 7: Helium performance compared to unoptimized and hand optimized code. We test the application runtime when pre-loading helium before and after manually optimizing the code for a set of four benchmarks, comparing against the non optimized version.

- *geoMatrix* computes the entrywise geometric mean of *n* matrices by computing *n-1* Hadamard product and a pointwise division. This process is present in many signal and image processing applications, such as lossy image compression algorithms. The baseline implementation composes the operations using generic binary functions. The hand optimized version uses specialized kernels for processing multiple matrices in-place and defines a tree reduction for the multiplication stage, exposing task parallelism though the OpenCL API.

- *ExpFusion*: this application [10] fuses several images taken with different exposure times into one to increase the dynamic range by using a Laplacian decomposition and a Gaussian pyramid. The depth of the pyramids, number of input images and their properties are not known statically, making the application highly dynamic. The hand optimized version makes some assumptions about the input to fuse some static parts of the pipeline (which can only apply to a subset of possible inputs).

Each application is tested with two different input sizes. The large input size is 4 times larger than the small one for all input and output buffers. For *geoMatrix* and *ExpFusion*, we also increase the number of inputs: *geoMatrix* is tested with 16 and 32 input matrices; *ExpFusion* with 4 and 12 input images and a pyramid depth of 4 and 8 respectively.

Each benchmark is executed ten times with and without preloading our framework, measuring the total wall clock time on the host between the first OpenCL action pushed in a command queue to the termination of the last synchronization or blocking operation. We report the analysis for the median point of the ten executions. As both the Nvidia driver and Helium use a persistent compiler cache, most of the compilation overhead is excluded from the measurements. However, the overhead of the analysis and task graph transformation is still present since the trace is regenerated with each execution.

The machine used for the test has an Intel Core i7-4770K CPU with 16GB of RAM and an Nvidia GeForce GTX 780 GPU connected via PCI-E 3.0. We use the OpenCL 1.1 implementation included in Nvidia's Linux driver 331.79. Since Helium's backend relies on Nvidia's open source PTX backend, we only evaluated the benchmarks on the GPU.

Table 1: **Performance impact of Helium on non-optimized baseline and hand-optimized version. For each application we report how many commands were issued, and how many of those were kernel invocations in parenthesis. All speedups are relative to the non-optimized code. The bold speedup numbers present the average speedup of both large and small inputs.**

	Unoptimized	Unoptimized + Helium		Hand Optimized		Hand Optimized + Helium	
Application	# Tasks Enqueued	# Tasks Replayed	Speedup	# Tasks Enqueued	Speedup	# Tasks Replayed	Speedup
CCO			**1.14**×		**1.49**×		**1.63**×
small	40 (10)	40 (10)	1.14×	80 (20)	1.50×	80 (20)	1.63×
large	40 (10)	40 (10)	1.15×	80 (20)	1.49×	80 (20)	1.63×
Sobel			**1.98**×		**1.98**×		**1.98**×
small	5 (3)	3 (1)	1.78×	3 (1)	1.80×	3 (1)	1.79×
large	5 (3)	3 (1)	2.17×	3 (1)	2.17×	3 (1)	2.17×
geoMatrix			**3.06**×		**1.68**×		**2.45**×
small	19 (18)	2 (1)	3.05×	10 (9)	1.68×	2 (1)	2.47×
large	35 (34)	2 (1)	3.06×	18 (17)	1.68×	2 (1)	2.44×
ExpFusion			**1.36**×		**1.04**×		**1.36**×
small	446 (441)	298 (293)	1.32×	339 (334)	1.05×	298 (293)	1.31×
large	2198 (2185)	820 (807)	1.41×	1727 (1714)	1.02×	820 (807)	1.41×

6. RESULTS

Figure 7 summarizes the effect of Helium on unoptimized and hand-optimized versions of our benchmarks. We notice that, Helium is able to improve performance over unoptimized code in all cases and further improve hand optimization in all cases except one where the performance is on par. Table 1 describes the experimental results in more detail. For each application and input, we report the number of OpenCL commands pushed in the command queue and how many of those were kernel invocations. We compare three alternative executions: the unoptimized binary with Helium preloaded, a manually hand-optimized version and lastly Helium with the hand-optimized binary. For each alternative execution we report the performance relative to the unoptimized application and the number of commands actually executed by the vendor implementation. The findings for each application are discussed below.

CCO. Helium is able to introduce task parallelism from the baseline using its parallelizing scheduler. Since there are no dependencies between the read at the end of an iteration and the writes from the following iteration, computation and communication can overlap. This results in a speedup of 1.15x. The hand optimized version does a better job by fragmenting the computation in smaller units, increasing the scope for copy-compute overlap, achieving 1.49x. However, combining Helium and the hand optimized version yields the best performance at 1.63x. Helium also takes advantage of the fragmented tasks but dispatches the tasks in three out-of-order queues; exposing three-way parallelism between computation and communication both from and to the device. Doing so manually requires a major re-write of the already hand-optimized code.

This shows that while Helium is able to improve the performance of existing code, it may also benefit from hand-transformations exposing more optimization opportunities.

Sobel. The first two stages being data independent and their output being processed by a map function, the three kernels can safely be merged into one, resulting in two fewer store instructions per point (for each temporary buffer) and three fewer loads per point (two for the temporary and one redundant read). When isolating this pattern in a single application, the same conclusion can easily be reached by a programmer, who applied the same optimization strategies in the hand optimized version. All implementations generated the same code and achieve the same speedup of 1.98x. However, if this pattern is part of a larger image application, or if it the result of dynamically composing from operators at runtime, it would become increasingly difficult to optimize by hand.

This application shows that Helium performs the same transformations as an expert programmer, without bloating the code base with specialized versions of the code.

geoMatrix. Because the number of input matrices is not known statically, it is not possible to implement a specialized version of the kernel. The hand optimized implementation achieves a speedup of 1.68x using in-place operations and creating specialized operators to multiply three matrices at once. Helium also generated specialized kernels, but combining all the matrices at once since their number is known at runtime. For both input sizes, Helium generated a single kernel, improving performance by over 3x.

While opting for a reasonable strategy of parallelizing tasks in the multiply stage, the manual transformations did not result in an important performance improvement. Task parallelism is not exploited by the GPU in this case because both inputs generate enough threads to occupy the entire device. However, this optimization considerably increases the complexity of the dataflow paths, resulting in poorer performance gains by Helium compared to Helium operating on the baseline. In both cases, our system performs the same optimizations: all kernels are fused into a single specialized kernel, and the amount of computation is roughly the same in both cases. The difference comes from the use of memory. By optimizing a composition in the baseline, the optimized version resulted in a single load per input matrix element and a single write for the result. The hand-optimized version uses writes to temporary buffers to speed up the reduction tree stage. These writes cannot be eliminated, as they are not released until after reading the final result. Hence, their side effect cannot be predicted and Helium cannot eliminate the dead store operations.

This demonstrates that partial hand-optimization can actually be counter-productive and impair performance if it is attempted prior to automatic techniques.

ExpFusion. Exposure fusion is a highly dynamic application and very little can be known statically. First, the pre-processing step depends on the image format, which is unknown at compile time. Second, the number of input images is unknown as well. A Laplacian Pyramid of an arbitrary depth is then built by recursively blurring and down-sampling the images, and the final image is reconstructed using a Gaussian pyramid. Despite the complexity of the pipeline, the host program only requires less than a hundred lines of code, but it contains complex control flow and requires a very modular design code to be maintainable.

The number of kernels executed at runtime widely varies depending on the input configuration, making manual optimizations very difficult, even with a help of a profiler. The small input size used 4 RGB input images and a pyramid depth of 4, which generated over 400 kernel instances. The larger input size used 12 input images and a depth of 8, invoking more than 2000 kernels.

The hand optimized version clones large parts of the code and specializes it for three-channels images, leaving the original generic application as a fallback if this assumption is not met at runtime. By specializing kernels, the overall number of invocations is decreased by 20%, however the performance gain is less than 5%, since most of the time is spent in combining across images rather than across image channels.

Helium takes advantage of runtime specialization to achieve the same speedup of 1.36x from either the baseline or the hand optimized code. It generates specialized code combining all images at once, leaving only the convolution steps in between, which cannot be merged due to spatial and temporal dependencies.

As the pyramid becomes wider and deeper with the larger input size, Helium shows better scalability than hand optimized code. It achieves a speedup of 1.4x, while the hand optimized version does not improve. This shows that even with more versions of the specialized kernels in the hand optimized implementation, Helium will alway stay ahead by generating them at runtime, allowing it to adapt to new inputs.

This last application demonstrates the applicability of Helium on large and dynamic workloads where hand optimizations are not applicable or not efficient.

7. RELATED WORK

Dataflow Analysis. Mistry et al. [11] developed a profiling technique for analyzing data flow in multi-kernel OpenCL applications. This approach does not apply optimizations but allows programmers to identify bottlenecks by manually inspecting a profiling trace. Jablin et al. [6] explored a CPU-GPU communication framework to track buffer usage and infer memory transfers between host and devices automatically. This scheme improves complex communication patterns but does not alter device code.

JIT Compiler Optimization. TaskGraph[3] is a C++ library for dynamic code generation. The computation is expressed in terms of nested components, from which a framework builds a high level computation AST, optimizes it at runtime and re-compiles the optimized version. Lancet[14] is a framework for interacting with the Java JIT compiler in order to specialize fragments of code. Instead of automatically infer dependencies and optimizations, the Lancet compiler relies on user annotations and explicit specialization, allowing users to control the transformations. The Java Virtual Machine uses a variety of runtime optimizers such as Jalapeño[2] and Graal[12], allowing aggressive dynamic transformations of Java programs.

Code Generator for Heterogeneous Systems. Code generators allow generation of highly tuned device code, often taking into account kernel sequences. Halide[13] is a language and compiler for implementing complex image processing pipelines. Their DSL provides high level scheduling API while the compiler supports many backends, including CUDA and OpenCL. StreamIt[5] is a language and a compiler for stream programs. The compiler analyses the resulting streams and performs optimizations such as task fusion and fission or reordering. Delite[4] is a framework allowing the user to implement domain specific languages and use a sophisticated compiler toolchain.

8. DISCUSSION AND FUTURE WORK

OpenCL implementations have demonstrated and popularized JIT compilation and JIT optimizations as a way to tame heterogeneity. The most recent version of OpenCL proposes interoperability between vendors at a much lower level with another standard, SPIR[8], allowing the kind of transformation applied by Helium to be performed even more efficiently and in a platform-independent way.

We showed that delaying OpenCL commands can be done completely transparently and that enough information can be gathered at runtime to drive more aggressive optimizations. As backend implementations get more efficient, lazily evaluated command queues might become the default behavior in OpenCL in order to maximize optimization potential, or at least become an alternative scheduling method proposed as an extension.

Helium can be used to integrate many other optimization techniques and is complementary to existing single-kernel compiler transformations. It would be beneficial for both single kernel and inter kernel optimizations to divide kernels in smaller atomic functions (kernel fission), which could then be re-assembled more effectively using Helium's task graph optimizer. This would allow Helium to factorize common memory operations and avoid redundant operations.

More research is necessary to evaluate and prioritize heuristics deciding how nodes should be fused and predict the efficiency of the transformation. Fusion is not always applicable since there are limitations on the number of kernel parameters. It is also not always beneficial for kernels using a lot of registers or for which the memory bandwidth has been optimized for a particular device. The challenges lie not only in predicting the performance gain of the transformation but also its re-usability in order to avoid over-specializing the code. Very aggressive specialization might not perform much better than a more selective specialization and is less applicable.

Finally the same technique can be adapted to distribute computation across multiple devices transparently. Analysis of a large task graph could find large independent sub-trees, which can be dispatched to different devices.

9. CONCLUSION

This paper has presented Helium, a transparent OpenCL overlay for automatically computing and optimizing task graphs in OpenCL applications. It is based on a delay-optimize-replay mechanism allowing the scheduler to chain OpenCL commands together according to their dependencies and compute only what is necessary in the host application. The optimizer also performs other types of transformations on the task graph, such as task reordering and kernel fusion, which improve the overall performance by increasing device occupancy and simplifying memory transactions across kernels.

We evaluated this framework on multiple benchmarks to assess both the efficiency of Helium's compiler transformations and its parallelizing scheduler. We found that in most cases Helium can replicate the performance of hand optimized code without the expense of refactoring code. For highly dynamic applications Helium can outperform hand optimized code by taking advantage of runtime information. Finally we showed that over-engineered and over-complicated code not only impairs maintainability but may also harm automated optimization processes, which could achieve better results with simpler code.

10. REFERENCES

[1] CUDA specifications. `http://docs.nvidia.com/cuda/`.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th Conf. on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2000.

[3] O. Beckmann, A. Houghton, M. Mellor, and P. Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*. 2004.

[4] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2011.

[5] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2002.

[6] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd Conf. on Programming Language Design and Implementation*, PLDI, 2011.

[7] Khronos OpenCL Working Group. *The OpenCL Specification version 1.2*, 19 edition, Nov. 2012. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[8] Khronos SPIR Working Group. *The SPIR Specification version 1.2*, 1 edition, Jan. 2014. `http://www.khronos.org/registry/spir/specs/spir_spec-1.2.pdf`.

[9] A. Magni, C. Dubach, and M. O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd Intl. Conf. on Parallel Architectures and Compilation*, PACT, 2014.

[10] T. Mertens, J. Kautz, and F. V. Reeth. Exposure fusion. In *Proceedings of Pacific Conf. on Computer Graphics and Applications*, 2007.

[11] P. Mistry, C. Gregg, N. Rubin, D. Kaeli, and K. Hazelwood. Analyzing program flow within a many-kernel opencl application. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU, 2011.

[12] OpenJDK. Graal project, 2013. `http://openjdk.java.net/ projects/graal`.

[13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th Conf. on Programming Language Design and Implementation*, PLDI, 2013.

[14] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision jit compilers. In *Proceedings of the 35th Conf. on Programming Language Design and Implementation*, PLDI, 2013.

[15] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2008.

[16] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.