`gpucc`: An Open-Source GPGPU Compiler

Jingyue Wu Artem Belevich Eli Bendersky Mark Heffernan Chris Leary
Jacques Pienaar Bjarke Roune Rob Springer Xuetian Weng Robert Hundt

Google, Inc., USA

{jingyue,tra,eliben,meheff,leary,jpienaar,broune,rspringer,xweng,rhundt}@google.com

Abstract

Graphics Processing Units have emerged as powerful accelerators for massively parallel, numerically intensive workloads. The two dominant software models for these devices are NVIDIA's CUDA and the cross-platform OpenCL standard. Until now, there has not been a fully open-source compiler targeting the CUDA environment, hampering general compiler and architecture research and making deployment difficult in datacenter or supercomputer environments.

In this paper, we present `gpucc`, an LLVM-based, fully open-source, CUDA compatible compiler for high performance computing. It performs various general and CUDA-specific optimizations to generate high performance code. The Clang-based frontend supports modern language features such as those in C++11 and C++14. Compile time is 8% faster than NVIDIA's toolchain (`nvcc`) and it reduces compile time by up to 2.4x for pathological compilations (>100 secs), which tend to dominate build times in parallel build environments. Compared to `nvcc`, `gpucc`'s runtime performance is on par for several open-source benchmarks, such as Rodinia (0.8% faster), SHOC (0.5% slower), or Tensor (3.7% faster). It outperforms `nvcc` on internal large-scale end-to-end benchmarks by up to 51.0%, with a geometric mean of 22.9%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization

General Terms Languages, Performance

Keywords GPU, compiler, optimization

CGO'16, March 12–18, 2016, Barcelona, Spain
ACM. 978-1-4503-3778-6/16/03...$15.00
http://dx.doi.org/10.1145/2854038.2854041

1. Introduction

Graphics Processing Units (GPU), such as NVIDIA's Tesla cards or AMD's FirePro cards, have emerged as powerful accelerators for massively parallel, numerically intensive workloads, which are often found in High Performance Computing (HPC). Recent advances in Machine Learning, specifically Deep Neural Networks (DNNs) and their applications, made GPUs appealing for large-scale datacenter operators, such as Google, Microsoft, Facebook, and Amazon. Similar to HPC codes, DNNs perform very large matrix-matrix and matrix-vector multiplies, as well as convolutions, which map well onto GPUs with their enormous computational power.

The two dominant programming models for GPUs are NVIDIA's CUDA [29] and OpenCL [30], which is supported on many platforms, including NVIDIA's. Both models are supported by vendor provided, proprietary development environments. While NVIDIA has previously open-sourced their NVPTX code generator [4] for LLVM, encouraging language and compiler development, there is currently no viable open-source CUDA compiler implementation that would approach performance and/or functionality of the proprietary CUDA toolchain.

As a result, almost no meaningful, reproducible research on compiler-based optimizations and productivity work for CUDA can be found. Many of the benefits that other open-source toolchains have brought to CPU environments cannot be materialized for GPU environments. Given the importance of these platforms and the size of the industry and academic communities, this appears to be a very large lost opportunity.

But this is not just a problem for general compiler research. Datacenter operators also try to avoid vendor-provided binaries as much as possible, specifically toolchains with their runtimes, for the following main reasons:

Security. Vendor-provided binaries might contain unintentional or, worst case, intentional security vulnerabilities, which could lead to leaks of high-value source code or personal identifiable information. Vendor-provided binaries pose these risks for both the build and deployment environments.

Performance. Compilers are tuned towards a common set of benchmarks, such as SPEC. However, these benchmarks might not be representative of important workloads in the datacenter. Being able to tune the compiler heuristics for specific workloads will inevitably lead to better performance. While these gains might be in the low percentage range, at the scale of datacenters or supercomputers every percent matters and can correspond to millions of dollars saved.

Binary dependencies. Allowing vendor-provided binaries into the datacenter would lead to massive binary dependencies – updating a core library would require that all dependent vendor binaries would have to be updated as well, all at the same time. This is intractable in general and impossible when vendors go out of business.

Bug turnaround times. Vendors serve many customers and have complicated release processes. Correspondingly, identifying and fixing a bug in the toolchain typically take from weeks to months. Having the compiler available in source can drastically speed up this turnaround time.

Modern language features. Languages like C++ evolve. Organizations might support more modern language standards or incompatible source environments which both might conflict with the subset of language features supported by proprietary compilers. This can and does make complicated source sandboxing necessary.

In this paper, we present gpucc, a fully functional, open-source, high performance, CUDA-compatible toolchain, based on LLVM [23] and Clang [1]. We developed, tuned, and augmented several general and CUDA-specific optimization passes. As a result, compared to `nvcc`, we are able to beat runtime performance for our end-to-end benchmarks by up to 51.0% with a geometric mean of 22.9% and on par with several open-source benchmarks. Compile time is 8% faster on average (geometric mean of compile times) and is improved by up to 2.4x on pathological cases (>100 secs). The compiler fully supports modern language features, such as C++11 and C++14. Our main contributions are:

- We developed and describe in detail `gpucc`, the first fully functional and completely open-source GPGPU compiler targeting CUDA.
- We identify and detail a key set of optimizations that are essential to generating efficient GPU code. With these optimizations, `gpucc` produces code that performs as well as or better than NVIDIA's proprietary compiler `nvcc` on a diverse set of public and internal benchmarks.

In the remainder of the paper, we describe the compiler architecture in §2, optimizations in §3, evaluation results in §4, and related work in §5.

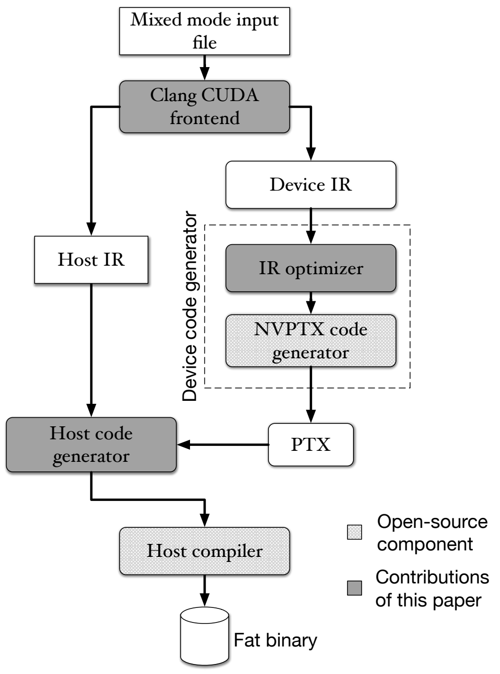

Figure 1: Overview of the compilation flow and execution of a mixed mode input file containing CPU and GPU code.

2. Overview

In this section, we will provide an overview of the system architecture and the compilation flow of `gpucc`.

The standard compilation of CUDA C/C++ programs consists of compiling functions that run on the device into a virtual ISA format dubbed PTX [28]. The PTX code is then compiled at runtime by the driver to the low-level machine instruction set called SASS (Shader ASSembler) that executes natively on NVIDIA GPU hardware.

One key design challenge for `gpucc` is to compile mixed code. Unlike the OpenCL programming mode which requires *host code* (code running on CPU) and *device code* (code running on GPU) to be separated, CUDA mixes host and device code in the same compilation unit (C++ source file) with a special invocation syntax (<<<...>>>) that allows host code to invoke device code. Compiling mixed code differs from traditional C++ compilation as two different architectures are targeted simultaneously.

2.1 Separate Compilation v.s. Dual-Mode Compilation

One possible approach to compile mixed-mode source files is *separate compilation*, which is adopted by `nvcc` and an early version of `gpucc`. This approach requires a component called the *splitter* that filters the mixed-mode code into source code required for the host (CPU) and the device (GPU). The splitter does this by performing source-to-source translation using Clang's tooling library [2]. The host

```
struct S {};

template <typename T>
__global__ void kernel(T *x) {}

template <typename T>
void host(T *x) {
  kernel<<<1, 2>>>(x);
}

int main() {
  S s;
  host(&s);
}
```

Figure 2: *A mixed-mode code example that demonstrates template instantiation across host and device boundary.*

and device source code are then compiled separately and finally merged into a fat binary.

However, separate compilation has two major disadvantages. First, the source-to-source translation it relies on is fragile. For example, C++ templates are a significant complication for splitting a mixed-mode input file. Declarations which should be compiled for the device are explicitly marked as `__device__` or `__global__` but emitting only functions and variables with these attributes is insufficient due to interdependencies between templatized host and device code. Specifically, templates can use compile time evaluated host functions to perform partial specialization, such as in Figure 2. As we perform source translation, these host functions would need to be retained. However, these retained host functions may use builtins (e.g., `__builtin_rdtsc()`) that are only available to the host. Therefore, the source-to-source translation needs to remove calls to device-specific builtins from the retained host functions.

The second disadvantage is that separate compilation requires unnecessary compilation time. Specifically, using this approach, `gpucc` has to parse source code four times – twice to filter the input into host and device source code and then twice when we compile them.

To overcome the above two disadvantages, the latest version of `gpucc` adopts the idea of *dual-mode compilation*. Instead of generating two intermediate source files, its frontend directly emits LLVM IR for the host and device avoiding the source-to-source translation. Also, template instantiation is no longer an issue because `gpucc` now has the complete translation unit which includes information about template instantiation on both host and device sides.

Figure 1 shows the architecture of `gpucc` with dual-mode compilation. The Clang CUDA frontend (§2.2) preprocesses and parses the input mixed-mode source file in two modes, one generating LLVM IR for the host and the other for the device. The resultant device IR is optimized and lowered to PTX assembly by the device generator (§2.3). Then, the host code generator (§2.4) injects the PTX to the host IR as a string literal. At runtime, the PTX assembly is JIT-compiled and executed by the CUDA driver.[1]

2.2 Clang CUDA Frontend

Given a mixed-mode input file, the Clang CUDA frontend (henceforth referred to as the frontend) parses the input file twice, once for host compilation and the other for device compilation. The generated host IR and device IR are then passed to the following host and device compilation.

To perform dual-mode parsing, the frontend needs to distinguish what code should be compiled for CPU and what should be compiled for GPU. The frontend does that leveraging function type qualifiers, a CUDA extension that indicates whether a function is executed on the host or device. A function declared with the `__host__` attribute or with no attributes (for compatibility with C++) is executed on the host. The `__device__` and `__global__` attribute declare a device function. The `__global__` attribute additionally declares a *kernel*, an entry to device code that can be launched via `<<<...>>>` from host code.

Unlike compiling a source file that contains only one language, the frontend also need to be aware of the differences between the CUDA and C++ languages; otherwise, when parsing the input for one target, the frontend would stumble on the code written for the other target. Below are the major differences between the two languages.

Target-specific predefined macros. A mixed-mode input can include headers that rely on target-specific macros predefined by the preprocessor, such as `__SSE__` defined on x86 and `__PTX__` defined on NVPTX. To preprocess these target-specific macros, `gpucc` simultaneously predefines both host- and device-specific macros. This approach allows the frontend to expand macros before its semantic analysis can distinguish what code is compiled for host or device. Defining both sets of macros simultaneously causes no conflicts, because CUDA and C++ agree on all overlapping macros, most of which are related to sizes and limits of data types.

The only macro that is predefined differently is `__CUDA_ARCH__`. In device mode, the frontend sets `__CUDA_ARCH__` to indicate the compute capability of the device targeted. This macro can be used by the programmer to generate different code sequences for different device (e.g., to use atomic instructions on GPUs that support it) as well as to distinguish between host and device compilation. For example, programmers use this facility to execute different code in `__host__ __device__` functions depending on whether the function is being executed on the host or device.

Compiler builtins. Some compiler builtins are target-specific. For example, `__builtin_rdtsc()` is only available during compilation targeting x86, and `__ldg()` is only avail-

[1] We are also experimenting compiling PTX to SASS before program execution and embedding the SASS directly into the resultant binary. This can save the run time spent on JIT compilation.

Callee / Caller	Host	Device	Global
Host	yes	no	only kernel launch
Device	last resort	yes	no
Global	last resort	yes	no

Table 1: *Requirements of function calls.* A function with `__host__ __device__` is treated as a host function in host mode and as a device function in device mode.

able on GPU. A mixed-mode input may use builtins that are specific for host and device targets, so the frontend needs to be aware of multiple compilation targets.

Language features. Some language features (e.g., thread-local storage, exceptions, and target-specific inline assembly) are not supported during device-side compilation, but the frontend still needs to be able to parse host code that uses them. Dual-mode compilation needs to suppress errors/warnings based on the target of the code where the issue is found. For example, when processing the input in device mode, the frontend will not produce any errors about thread-local storage variable used in a host function, even though they are not supported by the current target.

Function calls. Because functions can be executed on different targets, some function calls are disallowed. Table 1 shows what types of function calls are allowed and disallowed in host and device mode.

In general, callees that match current compilation mode are preferred, but it is possible for a device function to call a host function if there is no suitable alternative with a device attribute. This approach allows the device code to use a larger subset of host software without having to modify it to explicitly add target attributes. For example, the Thrust library calls functions declared in STL headers. If a device function calls a function that has both a device version and a host version, gpucc still prefers the device version as the callee.

2.3 Device Code Generator

The device code generator compiles the device IR generated by the frontend down to PTX. There are two submodules:

The LLVM IR optimizer runs a series of general and GPU-specific optimizations on the input IR, and passes the optimized IR to the NVPTX code generator. The IR optimizer heavily influences the quality of the generated PTX, and thus is crucial to producing high performing PTX for GPU applications. The specific optimizations performed by the IR optimizer will be discussed in §3.

The NVPTX code generator is the PTX code generator that was contributed to LLVM by NVIDIA. It translates LLVM IR to PTX which is compiled by NVIDIA's driver to SASS.

The PTX produced by the device code generator is injected as string literal constants into the host IR generated by the frontend, producing a "fat" binary containing both host and embedded device code.

```
__global__ void kernel(arg1, arg2) { ... }

int main() {
  ...
  kernel<<<grid, block>>>(arg1, arg2);
  return 0;
}
```

(a) *original input*

```
const char* __ptx =
    ".visible␣.entry␣kernel(\n"
    "␣␣.param␣.u64␣arg1,\n"
    "␣␣.param␣.u64␣arg2\n"
    ...;

__cuda_fatbin_wrapper = {..., __ptx, ...}

void __cuda_module_ctor() {
  __cudaRegisterFatBinary(__cuda_fatbin_wrapper);
  __cudaRegisterFunctions(kernel_stub, "kernel");
}

void kernel_stub(arg1, arg2) {
  cudaSetupArgument(arg1, ...);
  cudaSetupArgument(arg2, ...);
  cudaLaunch(kernel_stub);
}

int main() {
  ...
  cudaConfigureCall(grid, block, 0, nullptr);
  kernel_stub();
  return 0;
}
```

(b) *host IR*

Figure 3: *Host code generation.*

2.4 Host Code Generator

The host code generator injects the PTX to the host IR, and inserts CUDA runtime API calls[2]to load and launch the CUDA kernels in the injected PTX. Figure 3b shows an example of how the generated host code looks like. Note that the actual code is in LLVM IR; we show C++ here for clarity. The host code generator wraps the PTX in a global struct called `__cuda_fatbin_wrapper`. It also inserts a static initializer called `__cuda_module_ctor` that loads the PTX from `__cuda_fatbin_wrapper` and registers all the kernels using `__cudaRegisterFunctions`. For each CUDA kernel, it generates a kernel stub that prepares the arguments of the kernel (using `cudaSetupArgument`) and then launches the kernel. The only parameter of `cudaLaunch` is the address of the kernel stub which `__cuda_module_ctor` binds to the kernel name.

[2] Besides the CUDA runtime API, gpucc can target a new host runtime named StreamExecutor, which we will also open-source. StreamExecutor presents the user with a fluent C++ API with which to orchestrate data

3. Optimizations

This section describes some of gpucc's key IR-level optimizations. When we started building gpucc, it used LLVM's `-O3` optimization pipeline, which had been well tuned to CPUs but totally oblivious to the quirks of GPUs. Compared with our initial state, the optimizations described in this section resulted in a nearly 50x improvement in code performance. §4.3 will detail how each individual optimization impacts performance.

All these optimizations are implemented on and have been open-sourced to the LLVM compiler. Four of these optimizations and analyses (§3.2, §3.3, §3.4, and §3.5) are added as new passes in LLVM; two of them (§3.1) are existing optimizations tuned for CUDA programs; the remaining one (§3.6) is an existing optimization originally designed for other architectures and now enabled by us in gpucc.

3.1 Loop Unrolling and Function Inlining

Loop unrolling and function inlining are critical for gpucc's performance. Jumps and function calls are more expensive on common GPUs than on CPUs due to single-instruction multiple-thread execution, no out-of-order execution, and the pass-in-memory calling convention. Besides reducing jumps and function calls, loop unrolling and function inlining can expose more opportunities to Constant Propagation and Scalar Replacement of Aggregates (SROA) which promotes stack variables to registers. SROA removes the significant cost of accessing stack variables and has been observed to speed up some kernels by over 10x.

gpucc increases the function inlining threshold to 1100, which we select based on the overall effects on our benchmarks. The programmers can also use `#pragma unroll` and `__forceinline__` to force gpucc to unroll a loop and inline a function respectively. In our benchmarks, such annotations benefit both nvcc and gpucc.

3.2 Inferring Memory Spaces

CUDA C/C++ includes memory space designation as variable type qualifers (Table 2). Knowing the space of a memory access allows gpucc to emit faster PTX loads and stores. For example, a load from shared memory can be translated to `ld.shared` which is roughly 10% faster than a generic `ld` on an NVIDIA Tesla K40c.

Unfortunately, type qualifiers only apply to variable declarations so gpucc must infer the space of a pointer derived (via pointer arithmetic) from a variable. For example, although `p` in Figure 4 is not directly type-qualified, it can be proven to point to shared memory so programmers expect gpucc to emit `ld.shared.f32` for Line 5.

We address this challenge using an IR-level optimization called *memory space inference*. This optimization propagates the annotated memory spaces from variable declarations to its users. As a result, the NVPTX code generator can emit a load/store from a specific memory space if memory space inference proves that the address resides in that space.

Memory space	Attribute	Type qualifier
shared	`.shared`	`__shared__`
global	`.global`	`__global__`
local	`.local`	N/A
constant	`.const`	`__constant__`

Table 2: *Representation of memory spaces in PTX and CUDA.* Column **attribute** shows the memory space attributes PTX's `ld` and `st` instructions can optionally take. Column **type qualifier** shows the CUDA type qualifiers that programmers can use to annotate a variable residing in each memory space.

```
1  __shared__ float a[1024];
2  float *p = a;
3  float *end = a + 1024;
4  while (p != end) {
5    float v = *p;
6    ... // use "v"
7    ++p;
8  }
```

Figure 4: *A CUDA example that involves a pointer induction variable.* gpucc is able to prove that `p` in Line 5 points to shared memory.

We implemented memory space inference using a fixed-point data-flow analysis (Algorithm 1). gpucc runs `Propagate` on each function for each memory space MS. It first assumes all derived pointers (via pointer arithmetic) point to MS. Then, it iteratively reverts that assumption for pointers derived from another one that is not guaranteed in MS. In the end, GS and AS combined contains all pointers in the memory space MS.

This algorithm handles pointer induction variables which depend on themselves. For example, `p` in Figure 4 is initially assumed `shared`, and then remains `shared` because its incoming values are either `a` (guaranteed `shared`) or derived from `p` itself.

3.3 Memory-Space Alias Analysis

Besides enabling fast loads and stores (§3.2), knowing the space of a memory access also benefits alias analysis because memory spaces are logically disjoint. To leverage this feature, we add a memory-space alias analysis to gpucc that reports two pointers from different memory spaces as not aliasing. This new alias analysis makes dead store elimination more effective, and consequently speeds up the `lavaMD` benchmark in Rodinia by 5x.

transfers and computations. It supports both CUDA and OpenCL. Its CUDA support is built on top of the CUDA Driver API.

Algorithm 1: *Memory space inference.*

```
Input  : a function $F$ and a memory space $MS$
Output: a set of pointers guaranteed in $MS$
Propagate($F$, $MS$)
    $GS \leftarrow \emptyset$                                  // guaranteed in $MS$
    foreach pointer $P$ used in $F$ do
        // $P$ can be a global variable, argument, or
        instruction.
        if $P$ is guaranteed in $MS$ then
            $GS \leftarrow GS \cup \{P\}$
    $AS \leftarrow \emptyset$                                  // assumed in $MS$
    foreach instruction $I \in F$ that returns a pointer do
        if $I$ is derived from other pointers then
            $AS \leftarrow AS \cup \{I\}$
    while true do
        foreach $I \in F$ that does pointer arithmetic, pointer
        cast or PHI do
            foreach source $S$ of $I$ do
                if $S \notin GS$ and $S \notin AS$ then
                    $AS \leftarrow AS - \{I\}$
                    break
        if nothing has changed in this iteraion then
            break
    return $GS \cup AS$
```

3.4 Straight-Line Scalar Optimizations

This subsection describes gpucc's straight-line scalar optimizations. Their common feature is simplifying partially redundant expressions that perform integer or pointer arithmetic. For example, gpucc can rewrite `(b+1)*n` as `b*n+n` if `b*n` is already computed by its dominators.

These optimizations are particularly useful for HPC programs that often access arrays, e.g., matrix multiplication, dot product, and back propagation. These programs usually have unrolled loops (either unrolled manually by programmers or automatically by compilers) that walk through an array with a fixed access pattern. The expressions that compute the indices or pointer addresses of these accesses are often partially redundant. While we initially designed and implemented these optimizations for CUDA, some of them have been adopted by other backends in LLVM such as AMDGPU, PowerPC and ARM64.

Figure 6 shows a running example that we will use throughout this subsection. This example loads a 3x3 submatrix at indices `(b,c)` in the array `a`. The nested loops are fully unrolled to Figure 5a. Without straight-line optimizations, gpucc would emit very inefficient code by naïvely following the source code. For example, gpucc would compute `p8` in the as-is order, which is suboptimal for two major reasons.

- **Partial redundancy.** This computation order doesn't eliminate the partial redundancy between `(b+1)*n` and `(b+2)*n`. `(b+2)*n` could be replaced with `(b+1)*n+n` which takes only one extra add instruction.
- **Not leveraging addressing modes.** PTX ISA supports the `var+offset` addressing mode where `var` is a register and `offset` is a 32-bit immediate. If `p8` is only used by load/store instructions, we could reassociate `p8` to `(c+(b+2)*n)*4+8` so that the operation `+8` can be folded into addressing.

To attack these two sources of inefficiency, gpucc performs three major optimizations:

- *Pointer arithmetic reassociation* (§3.4.1) extracts constant offsets from pointer arithmetic, making them easier to be folded to the addressing mode `var+offset`.
- *Straight-line strength reduction* (§3.4.2) and *global reassociation* (§3.4.3) exploit partial redundancy in straight-line code and rewrite complex expressions into equivalent but cheaper ones. They together address the partial redundancy issue.

The end result of these optimizations is Figure 5d. Every `pi` except `p0` take at most one instruction. Even better, `p1`, `p2`, `p4`, `p5`, `p7`, and `p8` can be folded into addressing modes (thus free) if they are only used in loads or stores.

3.4.1 Pointer Arithmetic Reassociation

The goal of pointer arithmetic reassociation (*PAR*) is to fold more computation into addressing modes. Doing that is hard because the expression that computes an address might not be originally associated in a form that NVPTX's address folding matches.

In contrast to x86, which supports many sophisticated addressing modes, the only supported nontrivial addressing mode is `reg+immOff` where `reg` is a register and `immOff` is a constant byte offset. This addressing mode can be leveraged to fold pointer arithmetic that adds or subtracts an integer constant.

To leverage the addressing mode `reg+immOff`, PAR tries to extract an additive integer constant from the expression of a pointer address. It works by reassociating the expression that computes the address into the sum of a *variable part* and a *constant offset*. After that, the NVPTX codegen can fold the constant offset into the addressing mode `reg+immOff` via simple pattern matching.

Another benefit of PAR is that extracting constant offsets can often promote better Common Subexpression Elimination (CSE). For example, PAR transforms `p1` in Figure 5a to `&a[c+b*n]+1`, and CSE further optimizes that into `&p0[1]`. After that, the NVPTX codegen folds `p1` in a subsequent load instruction as `ld.f32 [%rd1+4]` where `%rd1` represents the value of `p0`. Figure 5b shows the end result of applying PAR and CSE to the running example.

```
                                                  x0 = b*n;
p0 = &a[c  + b   *n];  p0 = &a[c+b*n];            p0 = &a[c+x0];     p0 = &a[c+b*n];
p1 = &a[c+1+ b   *n];  p1 = &p0[1];               p1 = &p0[1];       p1 = &p0[1];
p2 = &a[c+2+ b   *n];  p2 = &p0[2];               p2 = &p0[2];       p2 = &p0[2];

                                                  x1 = x0+n;
p3 = &a[c  +(b+1)*n];  p3 = &a[c+(b+1)*n];        p3 = &a[c+x1];     p3 = &p0[n];
p4 = &a[c+1+(b+1)*n];  p4 = &p3[1];               p4 = &p3[1];       p4 = &p3[1];
p5 = &a[c+2+(b+1)*n];  p5 = &p3[2];               p5 = &p3[2];       p5 = &p3[2];

                                                  x2 = x1+n;
p6 = &a[c  +(b+2)*n];  p6 = &a[c+(b+2)*n];        p6 = &a[c+x2];     p6 = &p3[n];
p7 = &a[c+1+(b+2)*n];  p7 = &p6[1];               p7 = &p6[1];       p7 = &p6[1];
p8 = &a[c+2+(b+2)*n];  p8 = &p6[2];               p8 = &p6[2];       p8 = &p6[2];
```

(a) *after unrolling* **(b)** *after PAR+CSE* **(c)** *after SLSR* **(d)** *global reassociation*

Figure 5: *Straight-line scalar optimizations.*

```
#pragma unroll for (long x = 0; x < 3; ++x) {
  #pragma unroll for (long y = 0; y < 3; ++y) {
    float *p = &a[(c + y) + (b + x) * n];
    ... // load from p
  }
}
```

Figure 6: *The running example for straight-line scalar optimizations.*

`x = (b+C0)*s;`
`y = (b+C1)*s;`
	`x = b+C0*s;`
	`y = b+C1*s;`
		`x = &b[C0*s];`
		`y = &b[C1*s];`

`y = x+(C1-C0)*s`		

Table 3: *Strength reduction forms and replacements.*

3.4.2 Straight-Line Strength Reduction

Straight-Line Strength Reduction (*SLSR*) reduces the complexity of operations along dominator paths. It implements some ideas of SSAPRE [22] and handles more strength reduction candidates. The goal of SLSR is similar to that of loop strength reduction [14], but SLSR targets dominator paths instead of loops.

SLSR works by identifying strength reduction candidates in certain forms and making replacements for candidates in the same form. The current implementation of SLSR works on three forms shown in Table 3 where `b` and `s` are integer variables, and `C0` and `C1` are integer constants. For all the three forms, if `x` dominates `y`, SLSR can rewrite `y` as `x+(C1-C0)*s`.

For example, Figure 5b contains the following expressions that fit in Form 1 in Table 3.

```
x0 = b * n;
x1 = (b + 1) * n;
x2 = (b + 2) * n;
```

Applying the replacement rule, SLSR simplifies `x1` and `x2` to `x0+n` and `x1+n` respectively, reducing the cost of computing `x1` or `x2` from one add and one multiply to only one add. The end result of SLSR is Figure 5c, one more step closer to our desired end result (Figure 5d).

The transformations in Table 3 are likely beneficial but aggressive in some cases. The transformed code is likely better for two reasons. First, `C1-C0` is often 1 or -1, so `y=x+s` or `y=x-s` which takes only one PTX instruction to compute. Second, even if `C1-C0` is neither 1 nor -1, the cost of computing `(C1-C0)*s` is likely amortized because a loop usually accesses an array with a fixed stride. However, one caveat is that these transformations introduce extra dependencies and may hurt instruction-level parallelism. For NVIDIA's Tesla K40 GPUs, this issue is secondary to the number of instructions, because the hardware isn't pipelined and each warp has at most two integer function units running simultaneously. If needed, SLSR will perform some cost analysis to make smarter decisions.

3.4.3 Global Reassociation

Global reassociation reorders the operands of a commutative expression to promote better redundancy elimination. The idea is similar to the reassociation algorithm in Enhanced Scalar Replacement [13], but our algorithm is more compile-time efficient – its time complexity is linear in program size.

Global reassociation has the same goal as "local" reassociation [8] implemented in LLVM's `Reassociate` pass. However, the two approaches are different. Local reassociation transforms each expression individually according to some rankings that imprecisely reflect how values are associated in other expressions. In contrast, global reassociation transforms an expression according to how the operands are actually associated in its dominators. Therefore, global reassociation tends to select forms that expose more CSE opportunities. On the downside, global reassociation is more expensive because it makes decisions on global information.

The following program slice from Figure 5c demonstrates the benefit of global reassociation over local reassociation.

```
j0 = c + x0;
x1 = x0 + n;
j1 = c + x1;
```

Local reassociation favors grouping values that are not derived from other values, because non-derived values are more likely to be reused. Therefore, it chooses to reassociate `j1=c+x0+n` to `(c+n)+x0` since `x0` is derived from `b` and `n` (Figure 5c). This does not lead to further optimization in this case. Global reassociation instead chooses to associate `c` and `x0` first, because it observes that they are associated before in `j0`. The resultant code `(c+x0)+n` is further CSE'ed to `j0+n`, which costs only one add instruction.

Algorithm 2: *Global reassociation.*

```
Data: dominators($E$) maintains a list of observed
      instructions that compute $E$
Input  : the original program $P$ and a schedule $S$
Output: the specialized program
GlobalReassociation($F$)
    foreach instruction $I$ in pre-order of $domtree(F)$ do
        // + is used to represent any commutative operator.
        if $I = a + b$ then
            $E \leftarrow$ expr($I$)
            dominators[$E$] $\leftarrow$ dominators[$E$] $+ I$
        if $I = (a + b) + c$ and $a + b$ is used only once then
            $E_1 \leftarrow$ expr($a + c$)
            $D \leftarrow$ ClosestMatchingDom($E_1$, $I$)
            if $D \neq$ nil then
                Rewrite $I$ to $D + b$
            else
                $E_2 \leftarrow$ expr($b + c$)
                $D \leftarrow$ ClosestMatchingDom($E_2$, $I$)
                if $D \neq$ nil then
                    Rewrite $I$ to $D + a$
ClosestMatchingDom($E, I$)
    $D \leftarrow$ dominators[$E$]
    while $D \neq \emptyset$ and $\neg$dominate($D, I$) do
        popback($D$)
    if $D = \emptyset$ then
        return nil
    return back($D$)
```

Algorithm 2 shows the pseudo-code of our global reassociation algorithm for integer adds. It is dominator-based and runs in linear time to program size. The key data structure is a map called `dominators` that maps each expression E to a stack of observed instructions that compute E. The algorithm scans the instructions in pre-order of the dominator tree. Therefore, when it calls `ClosestMatchingDom` (E, I), all the dominators of I that compute E are already observed and present in `dominators`$[E]$. To find the closest dominator, the function `ClosestMatchingDom` keeps popping instructions out of `dominators`$[E]$ until the top of the stack dominates I. This pruning guarantees the algorithm runs in linear time, because every instruction is pushed or popped at most once; it is also safe because pre-order traversal guarantees that an instruction not dominating the current one will not dominate any instruction yet to be traversed either.

Similar to handling commutative integer operations, global reassociation also considers pointer arithmetic instructions. For example, Algorithm 2 transforms `p3` in Figure 5c to `&a[j0+n]`. Noticing that `&a[j0]` is already available as `p0`, global reassociation further simplifies `p3` to `&p0[n]` (Figure 5d).

```
                    p = &a[i];          p = &a[i];
                    if (b)              if (b)
if (b)                u = *p;             u = *p;
  u = a[i];         q = &a[i+j];        q = &p[j];
if (c)              if (c)              if (c)
  v = a[i+j];         v = *q;             v = *q;
```

(a) *original* **(b)** *speculative execution* **(c)** *straight-line optimizations*

Figure 7: *An example showing how speculative execution hoists instructions and promotes straight-line scalar optimizations.* Speculative execution transforms Figure a to Figure b, and then straight-line scalar optimizations further optimize it to Figure c.

3.5 Speculative Execution

One limitation of straight-line scalar optimizations (§3.4) is that they are unable to optimize instructions not dominating one or another (e.g., `&a[i]` and `&a[i+j]` in Figure 7a).

To address this limitation, we added an IR-level optimization to `gpucc` called *speculative execution*. It hoists side-effect free instructions from conditional basic blocks, so that they dominate more instructions and are consequently more likely to be optimized by straight-line scalar optimizations. For example, after speculative execution hoists `p=&a[i]` and `q=&a[i+j]` out of the conditional basic blocks (Figure 7b), `q` is dominated by `p` and can be optimized by global reassociation (§3.4.3) to `q=&p[j]` (Figure 7c). `gpucc` limits the number of instructions hoisted, and this threshold is currently selected based on the overall effects on our benchmarks.

Another benefit of speculative execution is promoting predication. Traditionally, compilers translate branches with conditional jumps. However, because jumps are expensive, common SIMD processors provide another execution model called *predication*. For example, all instruction on NVIDIA GPUs have an optional predicate. A predicated instruction is only executed when its predicate is true at runtime. With this support, NVIDIA's driver tends to translate a small conditional basic block to predicated instructions. This avoids jumps and the resultant straight-line code is more flexible for instruction scheduling. Speculative execution can hoist instructions from conditional basic blocks, rendering the remainder smaller and more likely to trigger predication.

3.6 Bypassing 64-Bit Divisions

Some of our internal end-to-end benchmarks frequently use 64-bit integer divides for computing array indices in GPU code. However, 64-bit integer divides are very slow on NVIDIA GPUs. NVIDIA GPUs do not have a divide unit so the operation is performed using a sequence of other arithmetic instructions. The length of this sequence increases superlinearly with the width of the divide. For example, a 64-bit divide requires approximately 70 machine instructions compared to approximately 20 machine instructions for a 32-bit divide.

Many of the 64-bit divides in our benchmarks have a divisor and dividend which fit in 32-bits at runtime. Leveraging this insight, `gpucc` provides a fast path for this common case. For each 64-bit divide operation `gpucc` emits code which checks whether the divide can be performed within 32-bits. If so, a much faster 32-bit divide is used to compute the result.

4. Evaluation

We evaluate `gpucc` on five key end-to-end benchmarks used internally and three open-source benchmark suites.

The five end-to-end benchmarks are from multiple domains of machine learning: `ic1` and `ic2` are for image classification, `nlp1` and `nlp2` are for natural language processing, and `mnist` is for handwritten digit recognition. These end-to-end benchmarks exercise a large amount of CUDA code. For example, `ic2` invokes 59 custom CUDA kernels.

The three open-source benchmark suites are:

Rodinia: 15 benchmarks from Rodinia 3.0. Rodinia [11] benchmarks are reduced from real world applications targeting GPU systems from multiple domains such as data-mining and medical imaging.

SHOC: 10 benchmarks from SHOC 1.1.4. SHOC [15] contains a diverse set of scientific computing benchmarks.

Tensor: 15 micro-benchmarks that exercise the Tensor [5] module in Eigen 3.0, an open-source C++ template library for linear algebra. This module contains over 18,000 lines of heavily-templatized CUDA code.

We omitted SHOC's `DeviceMemory`, `qtclustering`, `rnd`, and `spmv`, and Rodinia's `hybridsort`, `kmeans`, `leukocyte`, and `mumergpu`, because our implementation has not yet supported texture memory. We also excluded Rodinia's `backprop` due to licensing constraints.

Our evaluation machine has an Intel Xeon E5-2690 v2 and an NVIDIA Tesla K40c. The baseline compiler for our comparison is `nvcc` 7.0, the latest stable release of NVIDIA's commercial compiler at time of this writing.

In comparison to `nvcc`, the remainder of this section analyzes runtime performance (§4.1), compile time performance (§4.2), and the cumulative effects of the optimizations (§3) on the benchmarks (§4.3).

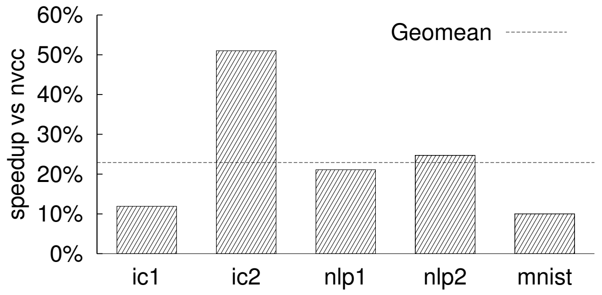

Figure 8: `gpucc` *vs* `nvcc` *on end-to-end benchmarks.* Each bar shows the relative speedup of the `gpucc` version of a benchmark. The higher the bar is, the better `gpucc` performs on that benchmark.

4.1 Runtime Performance Comparison

Emitting fast running GPU code is a key goal for `gpucc`. To evaluate the performance of `gpucc`-compiled code, we compile each benchmark using `gpucc` and `nvcc`, and compare the total GPU execution times (measured using `nvprof`) of both versions. For comparison, we measure relative speedup in percent using the formula $(\frac{\text{nvcc time}}{\text{gpucc time}} - 1) \times 100\%$.

End-to-end benchmarks. Figure 8 shows that `gpucc` generates faster code than `nvcc` on all the five end-to-end benchmarks by 10.0%-51.0% with a geometric mean of 22.9%. `gpucc` outperforms `nvcc` on these benchmarks mainly because of bypassing 64-bit divides (§3.6), which `nvcc` lacks, and better straight-line scalar optimizations (§3.4).

Open-source benchmarks. `gpucc` is on par with `nvcc` on open-source benchmarks. Figure 9 shows the performance comparison between `gpucc` and `nvcc` on Rodinia, SHOC, and Tensor. Overall, `gpucc` is on par with `nvcc` on all the three benchmark suites. The relative speedups range from -16.0% (for `algebraic` in Tensor) to 17.7% (for `convolution` in Tensor). The geometric mean relative speedups of Rodinia, SHOC, and Tensor are 0.8%, -0.5%, and 3.7%, respectively.

4.2 Compilation Time

Compile time is another important metric to measure `gpucc`'s quality. `gpucc` is 8% faster than `nvcc` on average (geometric mean of ratio of compilation time per translation unit) and significantly outperforms `nvcc` on the more complicated/template-heavy files. For example, one compilation unit from our benchmark suite takes 263.1s to compile with `nvcc` while `gpucc` takes 109.8s (2.4x faster compilation). In parallel build systems, the longest compile time of a file dominates the total compile time. The effect of reducing the compile time of the pathological cases therefore results in an even larger decrease in compilation time in parallel build systems and improves user experience.

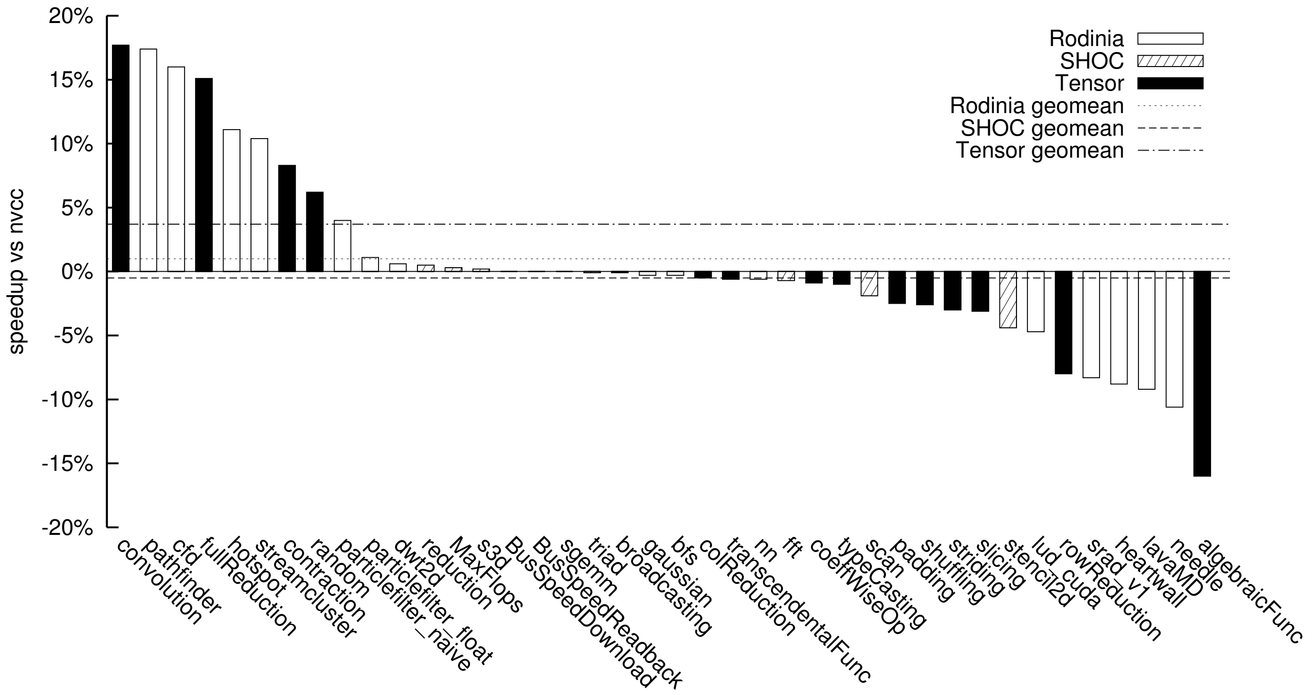

Figure 9: `gpucc` *vs* `nvcc` *on open-source benchmarks.* Each bar shows the relative speedup of the `gpucc` version of a benchmark. Bars with the same pattern represent benchmarks from the same benchmark suite. We also display the geometric mean speedup of each benchmark suite using a dashed horizontal line.

The compilation times reported are for `gpucc`'s separate compilation mode (§2.1). Separate compilation spends on average 34.7% of compile time performing splitting which will not be needed once the integration is done. Hence, `gpucc` users could get additional speedups in compilation time.

4.3 Effects of Optimizations

In this section, we describe the effects of `gpucc`'s optimizations (detailed in §3) on our evaluated benchmarks. This evaluation provides a guideline of how pervasive and impactful each optimization is in general.

For the end-to-end benchmarks, the most impactful optimizations are loop unrolling and function inlining (§3.1). Adding necessary source code annotations (such as `#pragma unroll`) and adjusting thresholds anecdotally sped up multiple benchmarks by over 10x. Other significant speedups include: memory space inference (§3.2) sped up `ic1` by over 3x; straight-line scalar optimizations (§3.4) sped up `ic1` by 28.3%; bypassing 64-bit divides (§3.6) sped up `ic2` by 50.0% and `nlp1` and `nlp2` by approximately 15%.

The effects on the open-source benchmarks are shown in Figure 10. We enabled the optimizations described in §3 one by one to evaluate the effect of each optimization. We merge speculative execution and straight-line scalar optimizations for this evaluation because they are inter-dependent (§3.5).

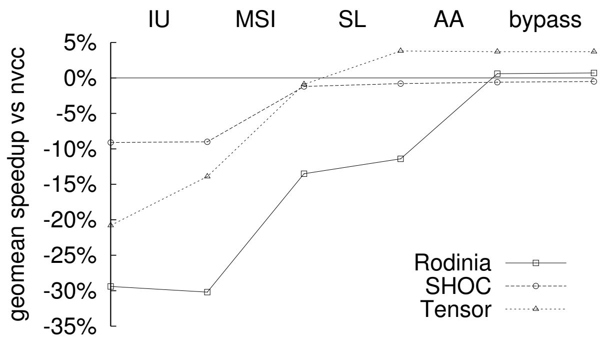

Figure 10: Cumulative effects on open-source benchmarks from increasing the inlining and unrolling thresholds (IU), inferring memory space (MSI), speculative execution and straight-line optimizations (SL), memory-space alias analysis (AA), and bypassing 64-bit divides (bypass).

All optimizations except bypassing 64-bit divides (§3.6) have an observable impact on at least one of the three benchmark suites. Bypassing 64-bit divides has no impact on these suites because they don't use 64-bit divides. Since the effects are measured as geometric-mean speedups across all benchmarks in a suite, the impact to individual benchmarks can be higher than the geometric mean. For example, although speculative execution and straight-line scalar optimizations speed up Tensor by an average of only 5%, they speeds up its `contraction` benchmark by 22.0%.

One interesting observation is that function inlining slightly regressed Rodinia. The reason for that is the `dwt2d` benchmark is register-intensive. While function inlining reduces the number of instructions, it can unfortunately increase the register pressure.

5. Related Work

We are unaware of other fully functional open-source compilers targeting CUDA. In the absence of such infrastructure, much of the GPU related architecture research has been conducted using simulators, such as GPGPUSIM [7]. The micro-architectural details for a given GPU architecture were determined via micro-benchmarking [33]. This type of analysis can inform compiler heuristics, but we did not use it in this effort.

Several compiler infrastructures address different abstraction levels, stages of the compilation, or attempt to cross compile to different target platforms. None of these allow compilation from CUDA to PTX. Ocelot [16] is a dynamic compiler which accepts PTX as input and uses LLVM as a backend and code generator to target several CPU platforms. As such, its input (PTX) is the output of our compiler. It does not perform the important high-level transformations during compilation from CUDA to PTX, and we therefore expect the resulting performance to remain sub-optimal. MCUDA [31] is a source-to-source compiler and a runtime component. It accepts CUDA kernels as input and translates them to run efficiently on multi-core CPUs. OpenMPC [24] is another source-to-source compiler for automatic translation of OpenMP applications to CUDA. POCL [21] accepts OpenCL as input and uses LLVM to target CPUs such as x86, Power, or ARM.

There have been several attempts at using higher-level domain-specific languages (DSLs) to generate CUDA kernels. Liquid Metal [6] translated input written in the Lime language and generates code for co-execution on GPUs, CPUs, and FPGAs. The Delite framework [9] uses embedded DSLs to target specific domains and generates code for GPUs and CPUs. Similar to the infrastructures mentioned above, these approaches do not target the full CUDA compilation flow and are unlikely to have as fine-tuned an optimization pipeline as described in this paper.

In regards to compiler optimizations, we find very few publications, and most of them appear to be based on source-to-source or PTX-level transformations to work around the unavailability of a functioning open-source CUDA compiler. Most optimizations target branch diversion and memory layout and placement, which are orthogonal to our optimizations. For example, Lee et al. [25] and Chakrabarti et al. [10] describes non-reproducible work done at NVIDIA that reduces branch divergence, analyzes memory spaces and vectorizes memory accesses. Han et al. [19] reduces branch divergence using iteration delaying and branch distribution which appear to be implemented as source-to-source transformations. Fauzia [17] uses instrumentation to find non-coalesced memory references and offers a PTX-level optimization technique. Porple [12] uses a small configuration language to specify memory placement of objects and combines it with an auto-tuner to achieve high performance.

Several publications address optimization work for different programming models and input languages. [20] describes Sponge, a compilation framework for streaming languages. [27] describes a C++11 compiler and runtime system that allows compiling C++ STL functions directly to GPUs. Similar work for C++14 is PACXX in [18]. Both of these C++ approaches don't detail optimizations and only evaluate microbenchmarks.

Some publications target optimizations at the CUDA source level. G-ADAPT [26] is an input-adaptive, cross-input predictive modeling technique that allows predicting near optimal CUDA program configurations. APR, a recursive parallel repackaging optimization technique is detailed in [34]. It operates at the CUDA source level and achieves impressive performance for their benchmarks.

The importance of SASS-level optimizations has been demonstrated by Tan [32], detailing non-reproducible work done at NVIDIA, and Nervana [3], showing the benefits of reverse engineered SASS. Both efforts target peak performance of matrix multiply kernels. We do not perform SASS-level optimizations, as the SASS specification is not publicly available.

6. Conclusion

We have presented `gpucc`, an open-source, high performance CUDA compiler. We have detailed its system architecture and optimizations that are essential to producing high-performing GPU code. Our results show that `gpucc` produces code that is faster than or on par with `nvcc`, has comparable or better compile times, and supports modern language features. The concepts and insights presented in this paper are general and apply, with modifications, to other architectures and programming models as well, such as OpenCL. We believe `gpucc` will enable meaningful, reproducible compiler and architecture research and can help make deployment of GPUs in restricted and controlled environments easier.

References

[1] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`, 2015.

[2] Libtooling. `http://clang.llvm.org/docs/LibTooling.html`, 2015.

[3] NervanaGPU library. `https://github.com/NervanaSystems/nervanagpu`, Mar. 2015.

[4] User guide for NVPTX back-end. `http://llvm.org/docs/NVPTXUsage.html`, Sept. 2015.

[5] Eigen 3.0 Tensor module. `http://bit.ly/1Jyh1FK`, 2015.

[6] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla. A compiler and runtime for heterogeneous computing. DAC '12, pages 271–276, 2012.

[7] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS 2009.*, pages 163–174, April 2009.

[8] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. PLDI '94, pages 159–170, 1994.

[9] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. PACT '11, pages 89–100, 2011.

[10] G. Chakrabarti, V. Grover, B. Aarts, X. Kong, M. Kudlur, Y. Lin, J. Marathe, M. Murphy, and J.-Z. Wang. CUDA: Compiling and optimizing for a GPU platform. *Procedia Computer Science*, 9:1910–1919, 2012.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09*, pages 44–54, Washington, DC, USA, 2009.

[12] G. Chen, B. Wu, D. Li, and X. Shen. PORPLE: An extensible optimizer for portable data placement on GPU. MICRO-47, pages 88–100, 2014.

[13] K. Cooper, J. Eckhardt, and K. Kennedy. Redundancy elimination revisited. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 12–21, 2008.

[14] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5): 603–625, Sept. 2001.

[15] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. GPGPU-3, pages 63–74, 2010.

[16] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. PACT '10, pages 353–364, 2010.

[17] N. Fauzia, L.-N. Pouchet, and P. Sadayappan. Characterizing and enhancing global memory data coalescing on GPUs. CGO '15, pages 12–22, 2015.

[18] M. Haidl and S. Gorlatch. PACXX: Towards a unified programming model for programming accelerators using C++14. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 1–11. IEEE Press, 2014.

[19] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in GPU programs. GPGPU-4, pages 3:1–3:8, 2011.

[20] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: Portable stream programming on graphics engines. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 381–392. ACM, 2011.

[21] P. Jääskeläinen, C. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. POCL: A performance-portable OpenCL implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.

[22] R. Kennedy, F. C. Chow, P. Dahl, S.-M. Liu, R. Lo, and M. Streich. Strength reduction via SSAPRE. CC '98, pages 144–158, 1998.

[23] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *PLDI '04*, pages 75–88, Mar. 2004.

[24] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC '10*, pages 1–11, 2010.

[25] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanović. Exploring the design space of SPMD divergence management on data-parallel architectures. pages 101–113, 2014.

[26] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for GPU programs optimization, 2008.

[27] T. Lutz and V. Grover. LambdaJIT: A dynamic compiler for heterogeneous optimizations of STL algorithms. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 99–108. ACM, 2014.

[28] NVIDIA. Parallel thread execution, ISA version 1.4.

[29] NVIDIA. CUDA programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, Mar. 2015. Version 7.0.

[30] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–73, 2010.

[31] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. Languages and compilers for parallel computing. chapter MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pages 16–30. Springer-Verlag, 2008.

[32] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast implementation of DGEMM on Fermi GPU. SC '11, pages 35:1–35:11, 2011.

[33] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS '10*, pages 235–246, March 2010.

[34] Y. Yu, X. He, H. Guo, S. Zhong, Y. Wang, X. Chen, and W. Xiao. APR: A novel parallel repacking algorithm for efficient GPGPU parallel code transformation. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 81:81–81:89. ACM, 2014.