Caracal: Dynamic Translation of Runtime Environments for GPUs

Rodrigo Domínguez
rdomingu@ece.neu.edu

Dana Schaa
dschaa@ece.neu.edu

David Kaeli
kaeli@ece.neu.edu

Department of Electrical and Computer Engineering
Northeastern University
Boston, MA

ABSTRACT

Graphics Processing Units (GPU) have become the platform of choice for accelerating a large range of data parallel and task parallel applications. Both AMD and NVIDIA have developed GPU implementations targeted at the high performance computing market. The rapid adoption of GPU computing has been greatly aided by the introduction of high-level programming environments such as NVIDIA's CUDA C and Khronos' OpenCL. Given the fact that CUDA C has been on the market for a number of years, a large number of applications have been developed in the HPC community.

In this paper we describe Caracal, our implementation of a dynamic translation framework that allows CUDA C programs to run on alternative GPU platforms. Here we target the AMD Evergreen family of GPUs. We discuss the challenges of compatibility and correctness faced by the translator using specific examples. We analyze the overhead of the translator compared with the execution time of several benchmarks. We also compare the quality of the code generated by our framework with that produced by the AMD OpenCL library. Our dynamically translated code performs comparably to the native OpenCL library, expands the opportunities for running CUDA C on new heterogeneous architectures, and provides a vehicle for evaluating compiler optimizations in the future.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); D.3.4 [**Programming Languages**]: Processors—*retargetable compilers, runtime environments*

General Terms

Design, Languages, Performance

Keywords

CUDA, CAL, OpenCL, Caracal, GPGPU, PTX, IL

To appear in the Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units, March 2011, Newport Beach, CA.

1. INTRODUCTION

GPUs have become the accelerator of choice in a number of key high performance applications. GPUs can obtain from 10x to 100x speedups in terms of execution time compared to CPUs. In 2006, NVIDIA introduced the CUDA C language[4] which lowered the threshold of entry to reap the benefits of GPU computing. CUDA C provides a C-like programming environment with simple extensions and a runtime environment to allow developers to execute non-graphics applications, called *kernels*, on the GPU. These accelerators are designed to exploit the data-level and task-level parallelism present in a wide range of applications.

As new architectures have emerged, the question of how to program many-core processors has become more critical. OpenCL[7] is an open standard maintained by the Khronos group that promises portability across different GPUs, operating systems, and multicore vendors. However, the present high performance programming environment is still under development, forcing OpenCL developers to tune their applications specifically to the underlying architecture.

Given the reality of ever-changing hardware and evolving language definitions, we need to move away from custom optimizations and utilize a more portable approach to GPU computing. To address this need, we focus on the interaction between the compiler and the underlying hardware. Current compilers for GPUs (i.e., shader compilers) need to be adapted to the new types of general purpose workloads, while maintaining the level of performance expected for the graphics pipeline.

In this paper we describe *Caracal*, an open-source framework that dynamically translates CUDA C programs compiled to the Parallel Thread Execution (PTX) virtual instruction set[8] to AMD's runtime environment for GPUs, called Compute Abstraction Layer (CAL)[1]. The main contributions of our work include:

- A framework that can be used by researchers to explore compiler optimizations and runtime techniques that map a range of programs to different CPU/GPU architectures.
- A comparison between two similar architectures and runtime environments that can be used in the analysis of GPU performance and programmability.

- A study of the challenges in dynamic translation and code generation when moving between these two massively parallel platforms.
- A description of the characteristics of the intermediate representations (IR) used by the compilation tools that can influence future designs.

The rest of this paper is organized as follows: Section 2 discusses related work and motivation behind Caracal; Section 3 describes the two architectures considered in this work; Section 4 presents the CUDA and CAL programming environments and compilation tools; Section 5 discusses the Ocelot framework used in Caracal and the extension we implemented for the CAL backend; Section 6 presents performance results and discusses the inherent challenges that still need to be overcome to obtain native performance with Caracal-translated binaries.

2. RELATED WORK

This section briefly discusses some of the previous work on translations and transformations targeted at GPU environments. In MCUDA[15], Stratton et al. propose source-to-source transformations to map CUDA C to multicore CPUs. They describe an algorithm to wrap kernel statements inside thread loops and perform loop fission to preserve the semantics of synchronization barriers. Their results show that the benefits of data locality and control structure expressed in CUDA C are transferable to CPUs.

Similarly, the Ocelot project[6, 9, 10] is a translation framework that can map CUDA C to x86 CPUs. Ocelot translates PTX to LLVM's IR[13] and then compiles it to run on the CPU. Ocelot is an open source project that is intended to provide a set of binary translation tools from PTX to several many-core architectures. It currently includes an internal representation for PTX, a PTX parser, a set of PTX to PTX transformation passes, a PTX emulator, and a dynamic compiler to many-core CPUs and NVIDIA GPUs.

The Twin Peaks project[11] presents an efficient implementation of the OpenCL standard for CPUs. In this work, the authors use `setjmp` and `longjmp` to provide barrier semantics on the CPU and discuss a mapping of the different address spaces in OpenCL to the CPU memory system taking into account cache locality.

To the best of our knowledge, Caracal is the first attempt to map CUDA C to AMD's GPUs and compare the two runtime environments. Our goal is to remove some of the barriers imposed when porting applications developed in CUDA C to non-NVIDIA platforms. This can be a boost to both the CUDA community, as well as to CPU and GPU vendors that are not presently targeted by CUDA.

3. HARDWARE ARCHITECTURE

This section describes the two GPU architectures discussed in the paper from the point of view of IR translation. Our discussion is focused on the AMD HD 5000 (Evergreen) and the NVIDIA GeForce 400 (Fermi) series. We start by identifying the similarities between the two architectures and proceed to highlight their differences. For clarity, we only discuss the architectural details that are relevant to the scope of this paper. Given that our framework is adaptable to other targets, we hope that by discussing our work on translation between the two most popular GPU platforms, others can use this model as a reference when considering how best to map CUDA C to their own architecture.

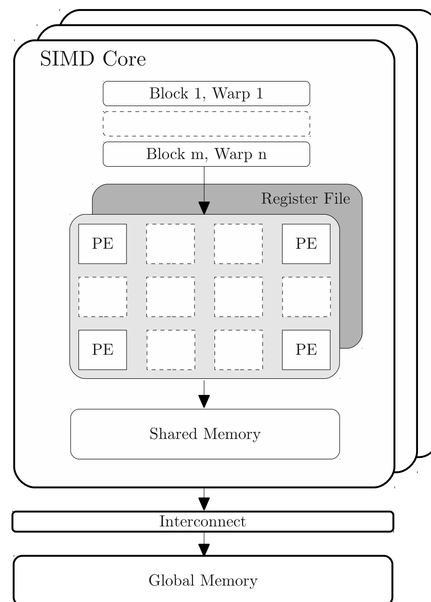


Figure 1: Hardware organization of a GPU.

GPUs architectures are designed to exploit the data-level parallelism available in a wide range of applications. GPUs achieve massive acceleration by dedicating most of the real estate on-chip to data processing (versus control flow and caches). In contrast to CPUs, which rely on caches to hide memory latency, GPUs have hundreds of threads that can be scheduled to hide any memory latency. GPU architectures are characterized by simple pipelines and in-order cores, eliminating the branch prediction logic or speculative execution capabilities typically found on most CPUs.

Figure 1 shows the hardware organization of a GPU. It consists of a scalable array of SIMD cores with several memory address spaces: (a) an off-chip memory connected via a high-bandwidth network (called *global memory*), (b) an on-chip software-managed scratchpad (called *shared memory*), and (c) a *register file* for local storage that is shared among the threads executing in the core.

The threads runnning on the GPU are organized in a hierarchical fashion, with different memory consistency models presented at each level. The smallest schedulable unit is called a *warp* (*wavefront* in AMD's terminology) which is a group of threads that share the program counter and, therefore, execute the same instruction in a lockstep fashion. The memory model at this level is sequential consistency.

At the next level, a group of warps forms a *block*. Threads belonging to the same block are guaranteed to execute on the same SIMD core and can communicate through the on-chip scratchpad for fast memory accesses. At this level, the memory model changes to a relaxed consistency model re-

Table 1: Hardware Specifications

Card	AMD HD5870	NVIDIA GTX480
SIMD Cores	20	15
PEs (per core)	16	32
Register File (KB)	256	128
Shared Memory (KB)	32	48
Global Memory (MB)	1024	1536
Memory Bandwidth (GB/s)	153	177
Core Clock (MHz)	850	1401
$Peak_{sp}$ (GFLOPS)	2720	1344
$Peak_{dp}$ (GFLOPS)	544	672

quiring explicit synchronization barriers.

Finally, a collection of blocks constitutes a *grid* whose threads are executed across the entire array of SIMD cores in no specific order and without synchronization. Threads from different blocks can only communicate through the off-chip memory using atomic instructions and memory fences. These hierarchical levels and consistency models give the hardware designers more flexibility and allow the architecture to scale easily. Note that GPUs are basically designed to perform graphics rendering, a highly data-parallel workload.

Each SIMD core is equipped with a number of Processing Elements (PE) to execute the threads. In AMD, each PE consists of a five-way VLIW pipeline and a branch execution unit[3, 16]. Every PE executes the same VLIW bundle from the current warp. Warps consist of 64 threads. Each PE can execute five single-precision multiply-add (MAD) operations per cycle and one double-precision MAD operation per cycle. Also, one of the VLIW lanes can be used to execute single-precision and double-precision transcendental operations. Efficient use of the AMD GPU relies on the hardware scheduler and the compiler to do vector packing to fill all five lanes of the pipeline.

Looking at the NVIDIA GPUs, each PE is a scalar pipeline that can execute one single-precision MAD operation per cycle and one double-precision MAD operation every two cycles[4]. Some of the PEs are special units that can execute single-precision transcendental functions. In contrast with AMD, NVIDIA's warp size is 32 threads.

The specifications for each GPU vary depending on the card. Table 1 shows an example of two cards introduced in the market around the same time and used in the experiments of this paper.

4. PROGRAMMING ENVIRONMENT

This section presents the CUDA and CAL programming environments and associated compilation tools. NVIDIA introduced CUDA C[4] to allow users to easily write code that runs on their GPUs. CUDA C consists of a simple set of extensions to C and a runtime library. Using CUDA C, developers can launch functions (referred to as kernels) to be executed on the GPU.

The programmer must partition the problem as a grid of blocks of threads when launching a kernel. Each thread can use intrinsics to identify its thread number as well as other launch configuration parameters. Figure 2a shows a kernel that adds two arrays `A` and `B` of `N` floats and writes the result to a third array `C`. Table 2 shows the address spaces exposed by CUDA and CAL. We omit the *local* and *texture* address spaces since they are outside of the scope of this paper.

Table 2: Address Spaces

CUDA	CAL
Global (with pointers)	Unordered Acess Views (with resources)
Shared	Local Data Share
Constant	Constant Buffers
Parameter	

Developers use the NVIDIA compiler `nvcc` to create an executable that includes the kernel in IR form. The executable calls the CUDA runtime (which in turn calls the driver) to compile the kernel IR into machine code and launch the application.

NVIDIA has defined an IR called PTX[8] that acts as a bytecode and allows the hardware to evolve without having to recompile legacy applications. This IR is just-in-time compiled by the driver, so only the driver needs to worry about the actual architecture in the system. Similarly, AMD defines an IR called Intermediate Language (IL)[2]. Figure 2b and Figure 2c show the PTX and IL corresponding to the vector addition example from Figure 2a.

Both PTX and IL allow for an unlimited number of virtual registers and defer the register allocation to the driver. This way most of the kernels are compiled directly into a Static Single Assignment (SSA) form making it easier for the driver to perform back-end optimizations.

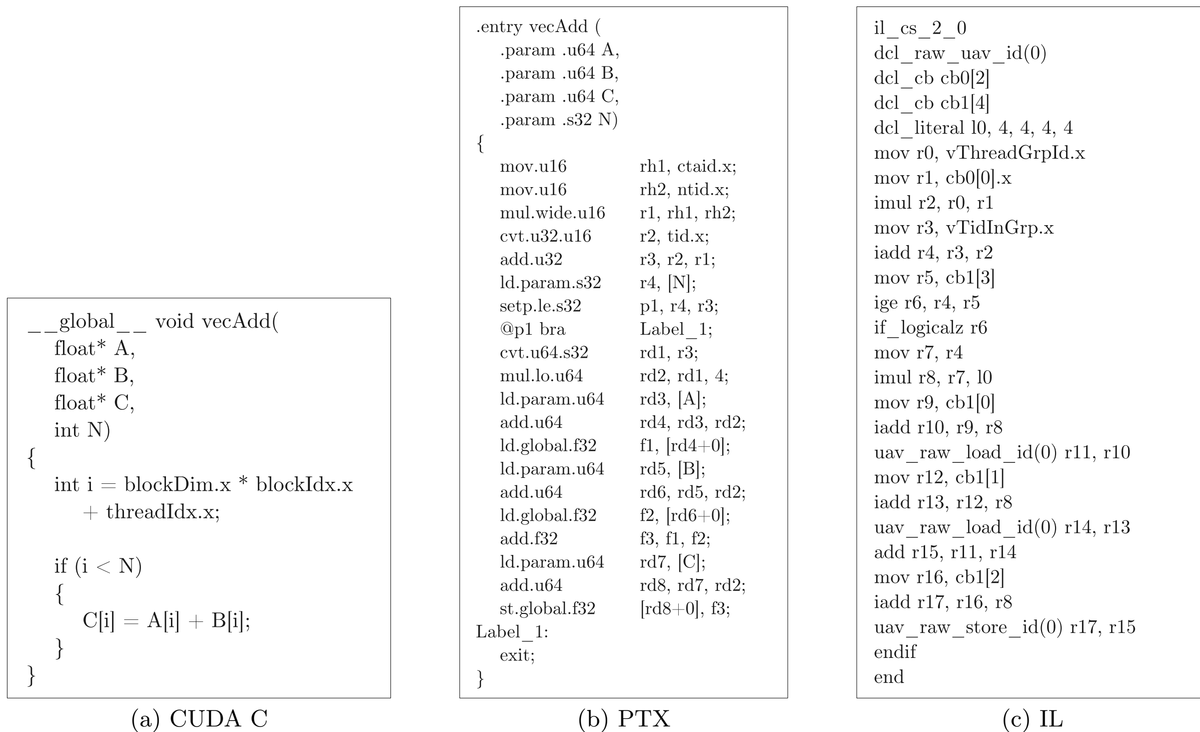
5. TRANSLATION FRAMEWORK

Caracal is based on the Ocelot translation framework for heterogeneous systems[6, 9, 10]. Ocelot allows CUDA C programs to run on NVIDIA GPUs and x86 CPUs. It implements the CUDA Runtime API[5] as a library that CUDA C programs can be linked to without any source modifications. It includes an internal representation for PTX, a PTX parser, a set of PTX to PTX transformation passes, a PTX emulator, and a dynamic compiler to many-core CPUs and NVIDIA GPUs.

In our work, we use the Ocelot front-end and develop a new IR targeted to AMD GPUs. This section describes the translation of the CUDA environment implemented in Caracal. We present the differences in the memory system, the IR formats, and the architectural details.

5.1 Global Memory

To implement global memory, we allocate a global buffer in CAL that we use as the global memory space and access it within the kernel using an Unordered Access View (UAV)[1]. We declare the UAV typeless (raw) in which case it is byte-addressable and memory operations must be doubleword-aligned. Additionally, we declare an Arena UAV for byte and short memory accesses and we bind it to the same buffer.



```
__global__ void vecAdd(
    float* A,
    float* B,
    float* C,
    int N)
{
    int i = blockDim.x * blockIdx.x
        + threadIdx.x;

    if (i < N)
    {
        C[i] = A[i] + B[i];
    }
}
```

(a) CUDA C

```
.entry vecAdd (
    .param .u64 A,
    .param .u64 B,
    .param .u64 C,
    .param .s32 N)
{
    mov.u16         rh1, ctaid.x;
    mov.u16         rh2, ntid.x;
    mul.wide.u16    r1, rh1, rh2;
    cvt.u32.u16     r2, tid.x;
    add.u32         r3, r2, r1;
    ld.param.s32    r4, [N];
    setp.le.s32     p1, r4, r3;
    @p1 bra         Label_1;
    cvt.u64.s32     rd1, r3;
    mul.lo.u64      rd2, rd1, 4;
    ld.param.u64    rd3, [A];
    add.u64         rd4, rd3, rd2;
    ld.global.f32   f1, [rd4+0];
    ld.param.u64    rd5, [B];
    add.u64         rd6, rd5, rd2;
    ld.global.f32   f2, [rd6+0];
    add.f32         f3, f1, f2;
    ld.param.u64    rd7, [C];
    add.u64         rd8, rd7, rd2;
    st.global.f32   [rd8+0], f3;
Label_1:
    exit;
}
```

(b) PTX

```
il_cs_2_0
dcl_raw_uav_id(0)
dcl_cb cb0[2]
dcl_cb cb1[4]
dcl_literal l0, 4, 4, 4, 4
mov r0, vThreadGrpId.x
mov r1, cb0[0].x
imul r2, r0, r1
mov r3, vTidInGrp.x
iadd r4, r3, r2
mov r5, cb1[3]
ige r6, r4, r5
if_logicalz r6
mov r7, r4
imul r8, r7, l0
mov r9, cb1[0]
iadd r10, r9, r8
uav_raw_load_id(0) r11, r10
mov r12, cb1[1]
iadd r13, r12, r8
uav_raw_load_id(0) r14, r13
add r15, r11, r14
mov r16, cb1[2]
iadd r17, r16, r8
uav_raw_store_id(0) r17, r15
endif
end
```

(c) IL

Figure 2: vectorAdd example.

Caracal handles dynamic global memory allocations (i.e. `cudaMalloc`) by managing the memory in the global buffer. We use a straightforward memory management algorithm. UAVs support atomic operations similar to the way they are handled in PTX.

5.2 Shared Memory

Shared memory is byte-addressable in both environments. However, PTX allows for shared memory accesses of different sizes: from one byte (e.g. `ld.shared.u8`) up to four doublewords (e.g. `ld.shared.v4.u32`). In IL, shared memory accesses must be aligned to a four byte boundary and the size must always be one doubleword.

Therefore, when translating a load operation from shared memory of size less than a doubleword, we need to extract the correct value from the result. The same applies for a store operation to shared memory where we have to merge the value with the contents in memory using the appropriate mask. This incurs an overhead of 7 IL instructions for loads and 13 IL instructions for stores. We plan to look at ways to reduce this overhead in future work.

PTX also provides variable declarations in shared memory (e.g. `.shared varname`). These variables can be used later as pointers (e.g. `mov r1, varname`). In IL, variables are not declared at all. In this case, the translator has to manage the layout of variables targeted for IL's shared memory space.

5.3 Constant Memory

Besides global and shared memory, PTX also defines a constant memory with an on-chip cache. CAL exposes a similar concept called Constant Buffers (CB)[1]. However, CBs are accessed in a non-linear way with each address consisting of 4 components (`x, y, z, and w`). This made translation of constant memory accesses non-trivial and for the initial implementation we mapped it to global memory (UAV). The downside of this approach is that we lose the cache capabilities of constant memory in Caracal. This remains an area of improvement in our framework.

5.4 IR Translation

In PTX, variables can be scalars or vectors of two or four components. However, only some instructions (`mov, ld, st, and tex`) allow vectors as operands. For all other instructions, vectors have to be unpacked into scalar variables.

On the AMD side, IL treats all variables as vectors of four components and all the instructions support vector operands. IL instructions have a rich set of modifiers to their operands including swizzles (e.g. `mov r0, r1.xxyy`).

Since we are translating from an environment that does not directly support vectors (versus AMD that support vectors operands for all instrucitons), this is not a problem. Our translator implements a straightforward solution mapping each PTX scalar variable to one of the components of an IL variable. Another option would be to apply vectorization techniques when translating the PTX scalar variables. This is an area of improvement that we plan to explore in the future. Note that since both PTX and IL allow for an unlimited number of virtual registers, our translation framework doesn't have to worry about register allocation nor spilling code.

One of the more challenging differences between PTX and IL is the control flow instructions. As shown in Figure 2, PTX is based on branch instructions and labels (e.g. `@p1 bra label`) while IL is based on structured constructs without labels (e.g. `iflogicalz-endif`). In order to perform correct translation and preserve proper program control flow, we need to identify the regions of the control flow graph (CFG) that match the constructs available in IL. Our implementation uses a technique called Structural Analysis[14, 12] which has been used before for data-flow analysis in optimizing compilers.

The goal of Structural Analysis is to build the control tree of the program – a tree-shaped representation where the leaf nodes are the individual basic blocks of the CFG and the rest of the nodes represent the constructs recognized by the target language. The algorithm works by doing a postorder traversal of the depth-first spanning tree, identifying regions that correspond to the structured constructs and reducing them into abstract nodes to build the control tree in the process. The algorithm stops when the abstract flowgraph is reduced to one node and this becomes the root node of the control tree. The translator can then iterate through the control tree starting from the root node and recursively process each one of the nodes.

It is possible that the algorithm will not match any regions in the graph to the associated IL constructs. In some cases, we can perform transformations (e.g. tail duplication) to turn the CFG into a reducible graph. The case of short-circuit evaluation of if statements in C/C++ is a good example. We are looking into techniques such as the one described in [17] to solve this kind of problem. Note that the amount of tail duplication could increase rapidly with the complexity of the graph. This also remains an open problem and an opportunity for future work.

In other cases, the graph could turn out to be irreducible. In otherwords, there is no way to represent the CFG using the constructs available in IL. In this case we are not able to translate the program without modifications to the source code.

5.5 Warp Size

Some applications are written to work with an specific warp size in mind. The warp size is used explicitly in the code to determine, for example, the dimensions of a data structure that is shared among the threads in the warp. We consider the implications of translating between two architectures with different warp sizes.

A good example to illustrate the potential impact of these differences is Histogram256 from the NVIDIA SDK. This program provides an efficient implementation of a 256-bin histogram. This application takes a set of values between 0 and 255 and finds the frequency of occurrence of each data element. In order to achieve good performance, the CUDA application divides the histogram into sub-histograms that are stored in shared memory and later merged into one.

Histogram256 must resolve memory collisions from multiple threads updating the same sub-histogram bin. However, this application was implemented to run on GPUs that did not have hardware support for shared memory atomics. Consequently, it uses the following code to implement a software version of an atomic operation in shared memory:

```
__device__ void addData256(
    volatile unsigned int *hist,
    unsigned int val,
    unsigned int tag)
{
    unsigned int c;
    do {
        c = hist[val] & 0x07FFFFFFU;
        c = tag | (c + 1);
        hist[val] = c;
    } while(hist[val] != c);
}
```

The application allocates a different sub-histogram to each warp and tags the bin counters according to the last thread that wrote to them. The parameters are a pointer to the per-warp sub-histogram, the value from the input set, and a tag that is unique to each thread in the warp.

A thread starts by reading the value of the counter and masking off the tag of the previous thread. The counter is incremented and the tag is replaced with the one from the current thread. Then, each thread writes back the new counter to the sub-histogram in shared memory.

In case of a collision between multiple threads from the same warp, the hardware commits only one of the writes and rejects the others. After the write, each thread reads from the same shared memory location. The threads that are able to write the new counter exit the loop. The threads involved in a collision whose writes are rejected execute the loop again. The warp continues execution when all the threads exit the loop.

It is important to allocate a different sub-histogram to each warp so there are no race conditions. Consider the following schedule involving two threads from different warps (indicated by the numbers on the left):

```
       Thread X                  Thread Y
  do {                      do {
1     c = hist[val] ...    4     c = hist[val] ...
2     c = tag | (c + 1)    5     c = tag | (c + 1)
3     hist[val] = c        6     hist[val] = c
8 } while(hist[val] != c)  7 } while(hist[val] != c)
```

If the threads were sharing the same sub-histogram and they were updating the same counter, this would lead to a race condition (thread X would increment the counter twice). Hence, it is necessary that threads from different warps do not share sub-histograms.

We are translating from an architecture with a warp size of 32 to a target with a warp size of 64. Fortunately, since 64 is a multiple of 32, we end up with two sub-histograms per warp, which is not a problem. However, it is worth noting

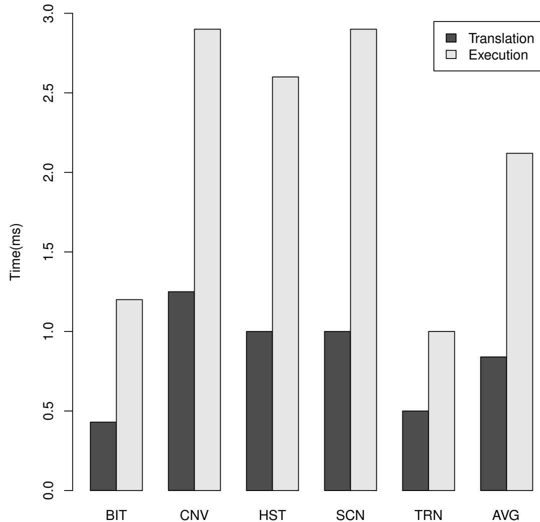


Figure 3: Translation vs Execution time.

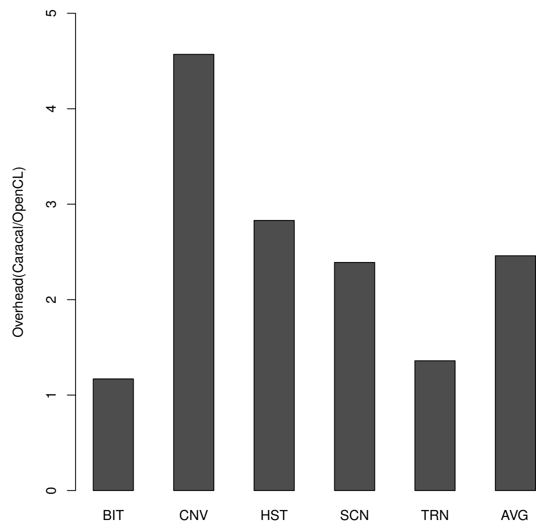


Figure 4: Performance evaluation of a kernel execution in Caracal versus OpenCL.

that if we were translating from a larger to a smaller warp size, this would have led to two warps sharing the same sub-histogram and resulted in a race condition.

6. EXPERIMENTAL SETUP

This section presents the evaluation of the Caracal translation framework. Our test platform is a 64-bit Linux (Ubuntu 9.10) system running on an Intel Core i7-920 CPU and an ATI Radeon HD5870 GPU. We ran the experiments using Ocelot (rev. 887), ATI Stream SDK 2.2, and the ATI Catalyst 10.9 driver.

The benchmarks are the applications from the CUDA SDK 2.3 shown in Table 3. We ported the CUDA benchmarks to OpenCL doing a one-to-one mapping of the language keywords and built-in functions and preserving the same optimizations on both version. We verified 100% functional correctness of our framework compared with CPU execution.

There are 3 major steps in launching a kernel in Caracal: the IR translation, the compilation to machine code (performed by the driver), and the actual execution of the application. Figure 3 compares the translation time against the execution time. The results show that, on average, the translation time is less than 1ms, which corresponds to approximately 1/3 of the execution time. Most of the translation time is spent doing the structural analysis of the kernel and building the control tree. Even though the translation is currently part of the launching process, it would be possible to implement an offline translator to avoid this overhead.

Figure 4 shows a comparison of the execution time of a kernel in Caracal versus OpenCL. The time measurements were taken using the `gettimeofday` system call in Linux around the kernel launch API (`calCtxRunProgramGrid` in Caracal and `clEnqueueNDRangeKernel` in OpenCL). Caracal performs comparably to OpenCL with a small overhead of 2.5x.

Table 4: Register Usage

Benchmark	Caracal	OpenCL
BIT	9	10
CNV	24	24
HST	14	14
SCN	7	6
TRN	4	4

The biggest overhead comes from Convolution (4.5x). This benchmark implements a separable convolution between a 2D image and a gaussian filter. The gaussian filter is declared in constant memory and is accessed by each thread inside a loop over the width of the filter. We credit the overhead to the fact that Caracal maps constant memory to a UAV (instead of a Constant Buffer) and, therefore, the filter does not get cached. In the future, we will evaluate the benefits of mapping constant memory to Constant Buffers in CAL.

As another metric of identifying code quality, Table 4 shows the per-thread register usage for both the native OpenCL kernels and the translated CUDA kernels. We collected the register usage from the AMD Stream Profile 1.4. All the kernels translated by Caracal have a register usage similar to that of the OpenCL library.

Convolution is the benchmark with the highest register usage. Given the level of register pressure, Convolution is limited by the number of thread blocks that can run on each SIMD core. This makes the application especially sensitive to memory delays since it lacks enough threads to hide the latency. This ratifies the performance overhead results presented in Figure 4.

Table 3: Benchmarks from the CUDA SDK 2.3

	Benchmark	Description	Working Set
BIT	Bitonic Sort	Sorting network of $O(n \log^2 n)$ comparators. Performs best when sorting a small number of elements.	256-integer array
CNV	Convolution	Separable convolution filter between a 2D image and a Gaussian blur filter.	3072×3072 image
HST	Histogram256	256-bin histogram calculation of an arbitrary-sized 8-bit data array.	100M-element array
SCN	Scan	Scan algorithm with $O(\log n)$ steps and $O(n)$ adds. Uses a balanced tree type algorithm (scan_best).	512-element array, 10K blocks
TRN	Transpose	Matrix transpose optimized to coalesce accesses to shared memory and avoid bank conflicts.	256×4096 matrix

7. CONCLUSIONS

In this paper we have presented the design of Caracal, an IR-level translation system based on Ocelot. We discussed many of the challenges faced when translating between the NVIDIA and AMD GPU runtime environments. We addressed issues of both compatibility and correctness encountered, and presented a prototype implementation that is functionally correct and performs comparably to the AMD OpenCL library with a small overhead of 2.5x evaluated over a set of benchmarks taken from the CUDA SDK. This framework can be used as a vehicle by other researchers to explore compiler optimization targeted to AMD GPUs.

8. ACKNOWLEDGMENTS

The authors would like to thank the members of the Ocelot mailing list, especially Gregory F. Diamos and Andrew R. Kerr, for their helpful discussions and their comments on our work. The work presented in this paper was supported in part by the NSF through an EEC Innovation Award (EEC-0946463), by AMD through the AMD Strategic Academic Partners Program, by NVIDIA through the NVIDIA CUDA Research Centers Program, and by support by the Vice Provost's Office of Research at Northeastern University.

9. REFERENCES

[1] ATI Compute Abstraction Layer (CAL) Programming Guide. `http://developer.amd.com/gpu/ATIStreamSDK/pages/Documentation.aspx`.

[2] ATI Intermediate Language (IL) Specification. `http://developer.amd.com/gpu/ATIStreamSDK/pages/Documentation.aspx`.

[3] ATI Stream SDK - OpenCL Programming Guide. `http://developer.amd.com/gpu/ATIStreamSDK/pages/Documentation.aspx`.

[4] CUDA C Programming Guide. `http://www.nvidia.com/object/cuda_get.html`.

[5] CUDA Reference Manual. `http://www.nvidia.com/object/cuda_get.html`.

[6] Ocelot website. `http://code.google.com/p/gpuocelot/`.

[7] OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[8] PTX: Parallel Thread Execution ISA. `http://www.nvidia.com/object/cuda_get.html`.

[9] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[10] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan. A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units*, 2011. To appear.

[11] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[12] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 171–185, New York, NY, USA, 1994. ACM.

[13] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[14] S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 7.7. Morgan Kaufmann, 1997.

[15] J. A. Stratton, S. S. Stone, and W. mei W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In *LCPC*, pages 16–30, 2008.

[16] J. Yang. AMD IHV Talk - Hardware and Optimizations. `http://sa09.idav.ucdavis.edu/docs/SA09_AMD_IHV.pdf`, Dec 2009. ACM SIGGRAPH ASIA 2009 Courses - OpenCL: Parallel Programming for Computing Graphics.

[17] F. Zhang and E. H. D'Hollander. Using hammock graphs to structure programs. *IEEE Trans. Softw. Eng.*, 30(4):231–245, 2004.