2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

# A portable and high-performance general matrix-multiply (GEMM) library for GPUs and single-chip CPU/GPU systems

Rahul Garg
School of Computer Science
McGill University
Montreal, Canada
rahul.garg@mail.mcgill.ca

Laurie Hendren
School of Computer Science
McGill University
Montreal, Canada
hendren@cs.mcgill.ca

***Abstract*—OpenCL is a vendor neutral and portable interface for programming parallel compute devices such as GPUs. Tuning OpenCL implementations of important library functions such as dense general matrix multiply (GEMM) for a particular device is a difficult problem. Further, OpenCL kernels tuned for a particular architecture perform poorly on other architectures.**

**We present a solution to the challenge of writing a portable and high-performance GEMM implementation. We designed and implemented RaijinCL, an OpenCL auto-tuning library for real and complex variants of GEMM that automatically generates tuned kernels for a given architecture. We comprehensively tested our library on a wide variety of architectures and show that the library is competitive with vendor libraries on all tested architectures.**

**We also implemented an autotuner for hybrid CPU+GPU GEMM that takes advantage of both the CPU and GPU on single-chip CPU+GPU platforms such as Intel Ivy Bridge. We show that our solution can outperform CPU-only, GPU-only as well as simple CPU+GPU tuning strategies. In addition to performance results, we provide analysis of architectural limitations as well as OpenCL compiler and runtime issues discovered on various systems, along with guidance on avoiding some of these issues.**

## I. Introduction

Hardware trends point to the rising importance of co-existence of a variety of compute devices such as multi-core CPUs, GPUs and DSPs. Considering only GPUs, different types of programmable GPUs are available from many different vendors, and each GPU family may have a very different architecture. Further, these GPUs may be integrated into the system in different ways. Some GPUs are discrete cards connected to the system via a bus like PCIe. Increasingly, GPUs are also being integrated on the same chip as the CPU and share some resources such as caches, memory interface and power budget.

OpenCL [1] is emerging as a common programming model and implementations are already available for many types of devices. In addition to a common programming model, users also want portable libraries. In this work, we focus on general matrix multiply (GEMM), which is the foundation of dense matrix operations libraries. On CPUs users rely on the BLAS (basic linear algebra subsystem) [2] application programming interface (API) for dense matrix operations. However, currently there is no standardized OpenCL BLAS API, nor any high-performance portable library.

Building a portable and high-performance OpenCL library for GEMM is challenging. Writing an OpenCL GEMM routine optimized for a single GPU family is itself a hard problem. However, writing a portable library is even harder because kernels tuned for one architecture usually perform poorly on other architectures. We provide a solution to this challenge. We have designed and implemented an OpenCL library called RaijinCL that automatically generates high-performance implementations of GEMM tuned for a particular architecture. The library achieves performance portability through auto-tuning, i.e. it tests various types of kernels and tuning parameters on each compute device in a user's machine, and caches the best found kernel for each compute device. The library delivers performance competitive with vendor BLAS on GPUs from multiple vendors.

Finding the optimal OpenCL kernels for the GPU is not enough for processors that integrate both the CPU and GPU on a single chip, such as recent processors from Intel and AMD. The user may want to distribute the computation over both the CPU and GPU to maximize the system throughput. The CPU and GPU may share resources such as memory bandwidth, power budget and caches. When the CPU and GPU are used together on a problem, there may be contention on some of these resources. Therefore, optimizing the system performance may require a different GPU kernel than the kernel best suited when GPU is acting on the problem alone. It is more than a scheduling problem, and instead requires finding a kernel and load distribution strategy co-tuned for the hybrid CPU/GPU case. We have extended our tuner to hybrid CPU/GPU systems and our library automatically determines the best system-level solution. Our solution significantly outperforms both CPU-only and GPU-only solutions, as well as naive CPU+GPU solutions.

The library is open-source[1]. RaijinCL provides implementations for GEMM, GEMV and transpose but here we focus on GEMM. We have received many queries about the library from both academic and commercial users, indicating that the library is of high interest to the community.

[1] http://www.raijincl.org

1066-6192/14 $31.00 © 2014 IEEE
DOI 10.1109/PDP.2014.40

### A. Contributions

We make five contributions in this paper.

**GEMM kernel design and search-space for autotuning:** We describe the design and implementation of our autotuning library including the set of kernels and parameter space explored by the library for GEMM. The details of our autotuner for GEMM is presented in Section II.

**Comprehensive performance evaluation showing good performance on all tested GPUs:** We present performance results from a range of architectures from multiple vendors on both real (SGEMM, DGEMM) and complex (CGEMM,ZGEMM) datatypes in Section III-A. We show that GEMM routines selected by our library achieve good performance in every case, and are competitive with the vendor's proprietary BLAS on various GPUs. Our study is the most comprehensive cross-vendor empirical comparison of GPU GEMM implementation that we are aware of in terms of vendor coverage as well as GEMM variation (SGEMM, DGEMM, CGEMM and ZGEMM) coverage.

**Analysis of GEMM on various architectures and compiler issues:** In addition to results, we present analysis of several aspects of the performance of the explored kernels on the tested GPUs. Our analysis includes both architectural aspects as well as several compiler issues found on various OpenCL implementations. This analysis and discussion of GPU results are presented in Section III-B.

**Hybrid CPU/GPU GEMM tuning and analysis:** We extended our autotuner to single-chip CPU/GPU systems. For the hybrid case, our autotuner tunes the GPU kernel and the load distribution between CPU and GPU at the same time. We tested single-chip CPU/GPU solutions from Intel and AMD and show that our hybrid approach delivers up to 50% better performance than CPU-only or GPU-only policies. We also show that the kernel tuned for a GPU-only policy is not necessarily the best kernel for the hybrid CPU/GPU case and thus it is not merely a scheduling problem. We also provide some guidelines that are critical to obtaining good performance on single-chip hybrid systems. We present our tuning methodology for maximizing system throughput in single-chip CPU/GPU systems in Section IV and results are presented in Section V.

**Intel Ivy Bridge platform results and analysis:** We are the first ones to cover Intel Ivy Bridge GPU architecture from the perspective of GEMM OpenCL implementation. We present a brief overview of the architecture and an analysis of our kernels on the architecture. In addition to performance results for both GPU-only and CPU+GPU performance, we also present data about power consumption of various strategies.

## II. General matrix-matrix multiply (GEMM): Autotuning and API design

RaijinCL provides implementations for GEMM, GEMV and transpose but here we focus on GEMM. We first provide an overview of the autotuning process and then discuss deployment and API design for GEMM. RaijinCL performs autotuning, i.e. it generates and tests many different kernels and selects the best one for a given device. RaijinCL implements a number of parameterized OpenCL kernel code generators called *codelets*. Each codelet represents a particular algorithmic variation of the solution. RaijinCL implements six codelets each for SGEMM, DGEMM, CGEMM and ZGEMM. For each parameter, we have a predefined list of possible values. The autotuner searches over all possible combination of parameters for each codelet. The best performing kernel and metadata (such as work-group size) required for execution of the kernel is cached. The tuning process is of the order of one hour to several hours depending upon the machine but only needs to be performed once for each OpenCL device and can be reused many times by any application using RaijinCL.

An important part of the design of RaijinCL was determining the correct codelets and parameters, so that the right design space is exposed for each kernel. We first describe the basic ideas in the design of the codelets and then describe the search space.

### A. GEMM Background

General matrix-multiply (GEMM) computes the operations $C = \alpha op(A)op(B) + \beta C$ where $A$, $B$ and $C$ are matrices, $\alpha$ and $\beta$ are scalars. $op(A)$ is either $A$ or $A^T$, and $op(B)$ is either $B$ or $B^T$ depending on input flags specified by the caller. Our GEMM API supports both row-major and column-major orientations. However, for this paper, we only consider row-major layouts. We support all four standard datatypes for the elements of $A$, $B$ and $C$: single-precision floating point, double-precision floating point, single-precision complex and double-precision complex which correspond to SGEMM, DGEMM, CGEMM and ZGEMM in the BLAS terminology.

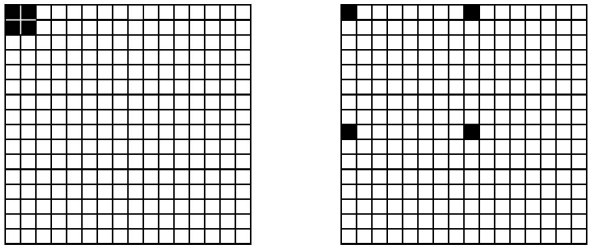
Four variations of GEMM can be considered based on whether the inputs are transposed : $C = \alpha AB + \beta C$, $C = \alpha A^T B + \beta C$, $C = \alpha AB^T + \beta C$ and $C = \alpha A^T B^T + \beta C$. These are called the NN, TN, NT and TT kernels respectively where N corresponds to no transpose and T corresponds to transpose. For square matrices the memory read pattern in TT kernel is very similar to the NN kernel, and we focus on NT, TN and NN layouts only for this paper. Let us assume we have an efficient kernel for any one of the three cases, and we have efficient transpose and copy routine. Then, one need not find efficient routines for the remaining layouts. One can simply transpose or copy the inputs appropriately, and then call the most efficient kernel. However, the memory access pattern for the TN, NT and NN cases can be quite different from each other and the layout found to perform the best on one architecture may not be the best layout on another architecture. Thus, we have implemented three variations of matrix multiplication: TN, NT and NN.

A naive row-major NN matrix-multiply kernel is shown in Listing 1. A naive OpenCL implementation will assign computation of one element of $C$ to one work-item. However, such an implementation will make poor use of the memory hierarchy of current compute devices. Thus, typically matrix multiplication is tiled. Each of the three loop directions $i$,$j$ and $k$, can be tiled with tiling parameters $T_i$, $T_j$ and $T_k$ and each work-item is assigned to compute a tile $T_i \times T_j$ of $C$. This tile is computed as a sum of a series of matrix multiplications of $T_i \times T_k$ and $T_k \times T_j$ tiles of $A$ and $B$ respectively.

```
int i,j,k;
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
    for(k=0;k<K;k++){
      C[i][j] += A[i][k]*B[k][j];
    }
  }
}
```

Listing 1: Row-major NN matrix-multiply

Consider a work-group of size $W_x, W_y$, where each work-item computes a tile of size $(T_i, T_j)$. The work-group will compute a $(W_x \times T_i, W_y \times T_j)$ tile of $C$. While we have specified the tile size, we have not specified how the tile-elements are assigned to work-items. We have implemented two possible assignments. The first assignment is that each work-item computes a tile consisting of $T_i$ consecutive rows and $T_j$ consecutive columns. The second possibility is that the $T_i$ rows are offset by $W_x$ from each other, and $T_j$ tiles are offset by $W_y$ from each other. We give a visual example. Consider a work-group of size $(8, 8)$ where each work-group is assigned $(2, 2)$ tile. Then, two possibilities for elements computed by the $(0, 0)$ work-item are shown in Figure 1.



(a) consecutive elements (b) offset elements

Fig. 1: Tiles computed by a work-item, consecutive or offset elements

### B. Codelets and search parameters

Our autotuner has six codelets for GEMM. Each codelet implements a particular layout (NT, TN or NN) and a particular element assignment scheme (consecutive or offset), thus giving six combinations. Each codelet has the following parameters:

*a) Local memory usage:* Each input matrix can be brought into a workgroup's local memory, which is shared across work-items, or fetched directly from global memory. Thus, there are two possible options for each input, and four possible values for this parameter.

*b) Use of OpenCL images:* Storing the input matrices as OpenCL images may be beneficial on some GPU architectures. Again, each input may or may not be stored as an image, and thus there are four possible values for this parameter.

*c) Tile sizes:* We explore tile sizes of 2, 4 and 8 for the number of rows and columns of output matrix. However, larger tile sizes increase register pressure. If register usage is more than the number of registers available, the compiler may spill registers which degrades performance substantially. Exploring such slow kernels will unnecessarily increase tuning time. Modern GPUs seem to have about 64 or 128 32-bit registers per thread. Thus, for each codelet, we estimate the register usage and prune out kernels that use more than 128 32-bit registers per work-item. For example, RaijinCL does not explore 8x8 tiles for DGEMM but does explore it for SGEMM. We also tile the inner loop and explore tile sizes of 1,2,4,8 and 16.

*d) SIMD width:* All loads and multiply-accumulate operations are done according to the value of this parameter. Our code generator explores SIMD widths of 1, 2 and 4.

*e) Work-group size:* The number of work-items inside the group. We currently search for two-dimensional work-groups with 32, 64, 128 and 256 work-items. We chose this set because most current GPU optimization manuals recommend work-group size be 32 or a multiple of 32.

The search space explored by the library for some of the parameters (such as tile sizes, register usage limit or work-group sizes) can be trivially changed by modifying constants in the code. Thus, other researchers are free to extend our experiments.

### C. Easy deployment

One issue with autotuning libraries is that deployment of the library can be hard. We have tried to simplify this task. RaijinCL itself can be distributed in binary form and comes with a small command-line utility for performing autotuning. The user specifies the device to tune for from the list of OpenCL capable devices installed on his/her machine. The tuning application only requires the OpenCL driver and does not require the user to install a C/C++ compiler. The tuning application generates and tests many different OpenCL kernels, and creates a device profile for the specified device. The device profile contains the best found OpenCL kernels as well as some metadata. The device profile is a simple self-contained text file and can be easily redistributed for deployment by other users of the same device. Device profiles for some popular devices like AMD Tahiti (which powers GPUs such as Radeon 7970) are available on our website. If a device profile is already available for your device, then you can simply download the profile and skip the tuning process completely. We are hoping that the community and hardware vendors will contribute many device profiles which will further simplify the deployment process.

### D. API design

*1) Asynchronous API:* Following OpenCL's design principles, RaijinCL's API is also asynchronous. Computationally heavy API calls in RaijinCL, such as GEMM, perform minimal setup and simply enqueue relevant kernel calls to the OpenCL device. Thus, RaijinCL API calls finish very fast without blocking the CPU and return an OpenCL event object.

*2) Control over resource allocation:* Efficient implementation of GEMM routines can require copying the arguments into a more efficient layout into a temporary buffer. Usually the GEMM library implementation will allocate, use and destroy the temporary buffers automatically without exposing them to the application programmer. RaijinCL offers both a high-level

API, that is similar to other GPU BLAS APIs such as AMD's OpenCL BLAS, and a low-level API. The high-level API implementation automatically allocates the temporary buffers, performs the appropriate copy or transpose operation and registers a callback with OpenCL to destroy the buffers when the kernel call is complete.

However, there are two performance related concerns with this high-level approach. First, consider a sequence of three matrix multiplication calls $A \cdot B$, $A \cdot C$, and $A \cdot D$. Here, the GEMM library will perform the allocate-copy-destroy operation for $A$ three times which is inefficient. Second, memory available to discrete GPU devices is often very limited and thus the application may want to explicitly manage the life-cycle of the memory objects allocated by the library. This is difficult to achieve in a high-level API. In the case of our high-level API, OpenCL runtime does not provide a guarantee of the amount of delay between the finishing of the kernel call and execution of the callback to destroy the buffers. Thus, in addition to the high-level API, we offer a four-step low-level API. First routine in the low-level API determines the size and type of temporary buffers required for a given input size and layout, and allocates them. The buffers may be reused for multiple problems of the same size and layout. Second, the programmer calls the copy routine. Third, the computation kernels are called. Finally, the temporary buffers can be deallocated. Thus, RaijinCL offers a choice between the convenience of a high-level API and control of a low-level API.

## III. Experimental results and analysis of GPU kernels

### A. Results

To evaluate our auto-tuning library we measured the performance of the kernels across 5 GPUs belonging to AMD GCN, AMD Northern Islands, Intel Ivy Bridge, Nvidia Fermi and Nvidia Kepler architectures. We report percentage of peak attained by RaijinCL and vendor BLAS (where available) on each architecture in the best case for each library in Table I.

| Architecture<br>Vendor<br>Type | GCN<br>AMD<br>GPU | N.Islands<br>AMD<br>GPU | Fermi<br>Nvidia<br>GPU | Kepler<br>Nvidia<br>GPU | IB GPU<br>Intel<br>GPU |
|---|---|---|---|---|---|
| SGEMM (RaijinCL) | 71.8 | 71.7 | 69.5 | 44.1 | 52.7 |
| SGEMM (vendor) | 64.9 | 66.2 | 69.0 | 45.1 | N/A |
| DGEMM (RaijinCL) | 83.5 | 85.3 | 57.6 | 91.9 | N/A |
| DGEMM (vendor) | 65.5 | 84.7 | 60.7 | 93.0 | N/A |
| CGEMM (RaijinCL) | 82.2 | 65.8 | 77.9 | 48.5 | 60.7 |
| CGEMM (vendor) | 65.6 | 64.5 | 80.8 | 51.9 | N/A |
| ZGEMM (RaijinCL) | 89.7 | 85.5 | 67.5 | 94.5 | N/A |
| ZGEMM (vendor) | 65.7 | 85.1 | 66.6 | 88.3 | N/A |

TABLE I: Percentage of peak obtained on each device in best case

The highlights are our results are:

- We have spanned all recent GPGPU architectures from all three major desktop vendors (AMD, Intel and Nvidia). We have also covered all four GEMM variations (SGEMM, DGEMM, CGEMM and ZGEMM). To the best of our knowledge, this work presents the most comprehensive study of GEMM covering all major recent desktop GPU architectures.
- The GEMM routine generated by our library outperformed AMD's OpenCL BLAS on both AMD GCN and AMD Northern Islands GPUs.
- Our library achieves about 97% of CUBLAS performance on real variants on both Nvidia Fermi and Kepler, and about 94% of CUBLAS performance on complex variants.
- On Intel's Ivy Bridge GPU architecture, we reached about 52% of peak on SGEMM and 60% of peak on CGEMM, which is higher than some architectures such as Nvidia Kepler. Intel does not provide a BLAS for Ivy Bridge GPU and thus our solution will be particularly important for users.
- Our autotuner found different parameter settings for all architectures, but found that TN was the best suited layout for all GPUs.

More detailed performance results for SGEMM, showing performance vs problem size, are shown in Figure 2. The machine configurations are detailed in Table II and optimal parameters found for SGEMM, DGEMM, CGEMM and ZGEMM are summarized in Table III.[2]

### B. Observations and analysis

**Best kernels on AMD architectures***:* On AMD's GCN and Northern Islands GPUs, each work-item in the fastest SGEMM kernels computed an 8x8 tile of the output matrix. Our kernel requires more than 64 32-bit registers for such a tile. Such kernels are not feasible on Nvidia's architectures discussed in this work because they have a limit of 63 registers per work-item. However, AMD's architectures do not have such limitations and thus are able to store such tiles entirely within registers. We found that the best performing kernels on GCN do not utilize local memory and do not copy data into images. Previously, some researchers, such as Du et al. [3], had noted that copying data into images is required for a high-performance GEMM implementation on some AMD architectures. The discrepancy between our result and previous research can be explained. Some AMD GPUs (such as Northern islands and previous-generation Evergreen) have a L1 data cache for read-only data which is quite important for performance on GEMM. Previously, this L1 cache was only used for image data but improvements in AMD's drivers now enable use of this cache for OpenCL buffers as well.

**AMD OpenCL compiler issues***:* We looked at the assembly code generated by AMD's OpenCL compiler for various kernels, and have identified two issues that may be preventing further performance improvements. First, register allocation is somewhat fragile. Even small changes in the code, such as commenting out unused variables, sometimes produced very different register allocations. Second, we also noticed that AMD's compiler reorders memory loads and does not follow the load order as written in the program. This prevents us from experimenting with the effect of various memory load orders. We are filing a request with AMD to provide a compiler option to disable this feature.

[2]We could not include plots of performance vs size for DGEMM, CGEMM and ZGEMM due to space constraints but these are available from our website or upon request.

| | GCN | N. Islands | Fermi | Kepler | Ivy Bridge |
|---|---|---|---|---|---|
| Vendor | AMD | AMD | Nvidia | Nvidia | Intel |
| GPU | Radeon HD 7970 | Radeon HD 8650G | Tesla C2050 | Geforce GT 650M | HD 4000 |
| OS | Kubuntu 12.04 (64-bit) | Windows 8 64-bit | Ubuntu 12.04 | Windows 7 64-bit | Windows 7 64-bit |
| CPU | Intel Core i7 3820 | AMD A10-5750 | Core i7 920 | Intel Core i7 3610QM | Intel Core i7 3610QM |
| RAM | 6GB DDR3 | 8GB DDR3 | 12GB DDR3 | 6GB DDR3 | 6GB DDR3 |
| Driver | Catalyst 13.4 | Catalyst 13.8b1 | 304.88 | 319.20 | 9.18.10.3071 |

TABLE II: Machine configurations

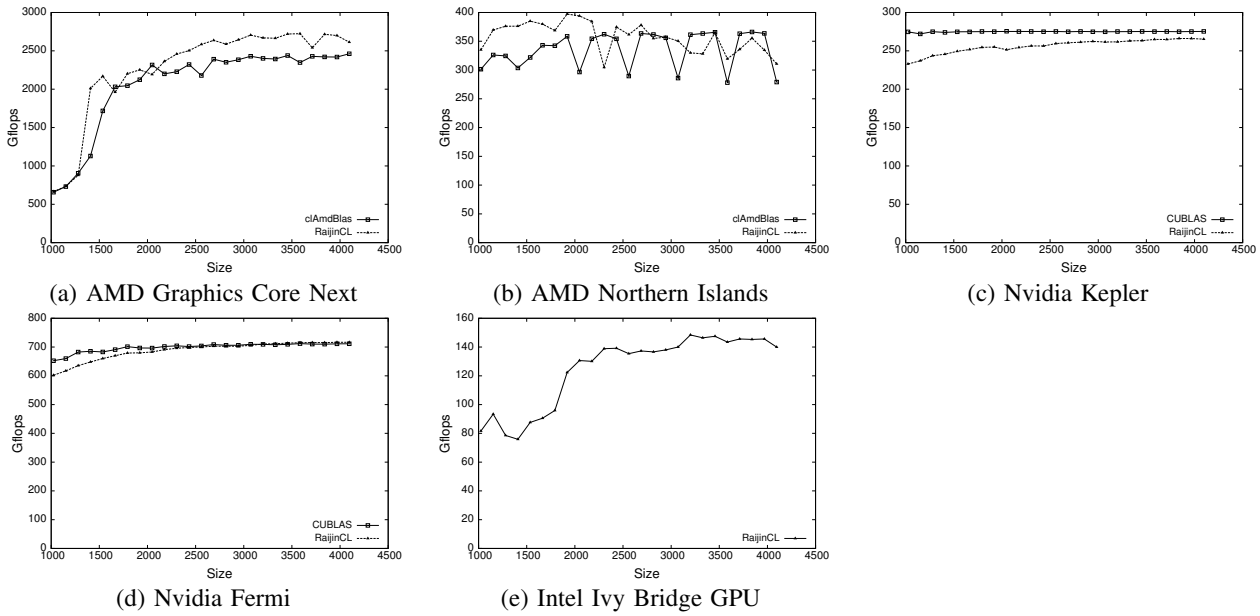
| Architecture<br>Vendor<br>Type | GCN<br>AMD<br>SGEMM | N.Islands<br>AMD<br>SGEMM | Fermi<br>Nvidia<br>SGEMM | Kepler<br>Nvidia<br>SGEMM | IB GPU<br>Intel<br>SGEMM | GCN<br>AMD<br>DGEMM | N.Islands<br>AMD<br>DGEMM | Fermi<br>Nvidia<br>DGEMM | Kepler<br>Nvidia<br>DGEMM |
|---|---|---|---|---|---|---|---|---|---|
| Layout | TN | TN | TN | TN | TN | TN | TN | TN | TN |
| Elements are consecutive | No | No | No | No | No | No | No | No | No |
| Tile size | (8,8,1) | (8,8,2) | (8,4,16) | (8,4,16) | (8,8,4) | (4,4,2) | (4,8,4) | (4,4,8) | (4,4,8) |
| Work-group size | (8,8) | (8,8) | (8,8) | (16,16) | (4,8) | (16,8) | (8,8) | (8,8) | (8,8) |
| SIMD width | 4 | 4 | 4 | 2 | 4 | 2 | 2 | 2 | 2 |
| Bring A to local memory | No | No | Yes | Yes | Yes | No | No | Yes | Yes |
| Bring B to local memory | No | No | Yes | Yes | Yes | No | No | Yes | Yes |
| Store A in image | No | No | Yes | Yes | No | No | No | Yes | Yes |
| Store B in image | No | No | Yes | Yes | Yes | No | No | Yes | Yes |

(a) SGEMM and DGEMM

| Architecture<br>Vendor<br>Type | GCN<br>AMD<br>CGEMM | N.Islands<br>AMD<br>CGEMM | Fermi<br>Nvidia<br>CGEMM | Kepler<br>Nvidia<br>CGEMM | IB GPU<br>Intel<br>CGEMM | GCN<br>AMD<br>ZGEMM | N.Islands<br>AMD<br>ZGEMM | Fermi<br>Nvidia<br>ZGEMM | Kepler<br>Nvidia<br>ZGEMM |
|---|---|---|---|---|---|---|---|---|---|
| Layout | TN | TN | TN | TN | TN | TN | TN | TN | TN |
| Elements are consecutive | No | No | No | No | No | No | No | No | No |
| Tile size | (4,4,2) | (4,4,2) | (4,4,8) | (4,4,16) | (4,4,8) | (4,4,1) | (4,4,2) | (2,4,8) | (4,2,1) |
| Work-group size | (8,8) | (8,8) | (8,8) | (8,16) | (8,8) | (8,8) | (8,8) | (8,8) | (16,4) |
| Complex vector size | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 |
| Bring A to local memory | No | No | Yes | Yes | Yes | No | No | Yes | No |
| Bring B to local memory | No | No | Yes | No | Yes | No | No | Yes | No |
| Store A in image | No | No | Yes | No | No | No | No | Yes | No |
| Store B in image | No | No | Yes | Yes | No | No | No | No | No |

(b) CGEMM and ZGEMM

TABLE III: Optimal Parameters for SGEMM, DGEMM, CGEMM and ZGEMM




(a) AMD Graphics Core Next (b) AMD Northern Islands (c) Nvidia Kepler

(d) Nvidia Fermi (e) Intel Ivy Bridge GPU

Fig. 2: GPU throughput versus problem size for SGEMM

**Nvidia Kepler local memory limitations***:* We found that generally loading both operands into local memory was desirable for both Fermi and Kepler architectures on most GEMM variations. However, on CGEMM, we found that the best kernel for Nvidia Kepler used local memory for only one of the inputs. Attempting to load tiles of both input matrices into local memory increased local memory requirements per work-group, decreasing the number of active work groups on each Kepler core. Decreased occupancy led to worse performance. Compared to Nvidia's prior generation Fermi, Nvidia increased the number of ALUs from 32 to 192 per core, while keeping the amount of local memory per core (48kB) the same. The limited size of the local memory on Kepler appears to be holding back the ALUs in this case.

**FMA instructions***:* OpenCL supports FMA as a built-in function. On supported architectures, FMA built-in maps to a hardware instruction while some others implement it in software. OpenCL provides a preprocessor macro FP_FAST_FMAF for OpenCL kernels that is defined when the implementation supports a fast FMA operation. However, we discovered that some implementations do define the macro, but performance sometimes varies with datatype (float vs double) or even SIMD width. To avoid such issues, RaijinCL runs some micro-benchmarks and attempt to automatically determine whether or not to use FMA for a particular datatype and SIMD width.

**Intel Ivy Bridge GPU architecture analysis***:* The Intel Ivy Bridge GPU (HD 4000) architecture is not well known in high performance computing related literature, and thus we provide a brief description. The HD 4000 GPU consists of two *sub-slices*. Each sub-slice has eight ALUs called execution units (EUs), each of which can perform 16 flops/cycle. Thus, the whole GPU can perform 256 flops/cycle. Each sub-slice has access to 64kB of local memory which is divided into 16 banks and provides 64 bytes/cycle of bandwidth.

OpenCL workgroups are mapped to a sub-slice and each sub-slice runs more than one workgroup. OpenCL workgroups are broken into EU threads, which are analogous to warps on Nvidia architectures and wavefronts on AMD architectures. EU threads are then distributed across EUs in the sub-slice. Each EU can run up to 8 EU threads to hide latencies and increase utilization similar to AMD and Nvidia architectures. However, unlike say Nvidia architectures with a fixed warp size, the number of work-items in a EU thread is kernel-dependent. Each EU thread has access to a 4kB register file. The compiler maps either 8, 16 or 32 work-items to one EU thread depending upon register usage of each work-item. We estimate that the best found kernel by RaijinCL for SGEMM on Ivy Bridge is using more than 64 registers, making it impossible to map 16 or 32 work-items to one EU thread. Thus, likely 8 work-items of our kernel are mapped to each EU thread and our work-group (32 work-items) is divided into 4 EU threads. For full occupancy, we need 64 EU threads (or 8 work-groups in our case) per sub-slice. Our kernel is using less than 2kB of local memory per workgroup, and thus there is enough local memory to run more than 16 work-groups and there should be enough EU threads to fully occupy all EUs.

RaijinCL achieved about 52% of peak on SGEMM (Figure 2(e)) and 62% on CGEMM. Our estimate is that there is enough local memory bandwidth to achieve about 85% of peak on our SGEMM kernel, compared to 52% achieved, and the bottleneck is likely elsewhere. We performed an additional experiment to see if global memory access is one of the bottlenecks. We removed all global memory loads from the auto-generated kernel and replaced the load value with just the address computation. The SGEMM performance improved to about 60% of peak, which supports the hypothesis that global memory access is indeed one of the bottlenecks.

## IV. Tuning for single-chip CPU+GPU systems

In the previous section, we looked at performance of GPUs in isolation to the rest of the system. In this section, we extend the discussion to system-level performance. System level performance considerations include performance of both the CPU and GPU, as well as data transfer and synchronization overheads between the CPU and the GPU. In single-chip CPU+GPU solutions such as Intel's Ivy Bridge and AMD's Richland APUs, the CPU and the GPU may share resources such as the memory controller and caches. Such chips also integrate sophisticated power management techniques where the CPU may boost beyond its boost clocks on CPU-heavy workloads when the GPU is idle, and the GPU may boost on GPU-heavy workloads when the CPU is idle.

We want to utilize both the CPU and the GPU where possible. Consider an output matrix $C$ of $M \times N$ elements. We logically partition the $M$ rows into $P$ parts of $M/P$ rows each. We assign the computation of $p_g$ parts out of $P$ to GPU, and $p_c$ parts out of $P$ to CPU. Here $p_g + p_c = P$ and thus determining $p_g$ automatically assigns $p_c$ for a fixed value of $P$. The value of $p_g$ chosen affects the net system performance. For a given value of $p_g$ and $P$, and a given auto-generated GPU kernel, we follow the following algorithm for computing an output matrix $C$ of size $M \times N$.

1) Copy the relevant input data to GPUs if required. The creation of GPU input buffers can be optimized on some single-chip systems that allow creation of GPU buffers without data copy.
2) Enqueue a call to the provided GPU kernel to compute $M * p_g/P$ rows of C.
3) Flush the GPU command queue to force the OpenCL runtime to begin execution of the kernel.
4) Call the CPU BLAS to compute $M * (P - p_g)/P$ remaining rows of C.
5) Enqueue a copy of data from the GPU to the final output matrix C. We are using in-order queues, so this call will only execute once the GPU computation is finished thus ensuring correctness.
6) Wait on the GPU command queue to finish execution of all GPU operations.
7) Delete all GPU buffers and free any other temporary resources.

In such a system, the net system performance depends on the selected GPU kernel as well as the parameter $p_g$. We have fixed $P$ to eight in our system as it offers a good balance of search space granularity and tuning time. We wanted to determine the best possible GPU kernel and $p_g$ parameters in a hybrid setting. The GPU kernel that provides the best performance when considering GPU performance in isolation may not be the best kernel when considering whole system

performance. Thus, the tuning for hybrid CPU/GPU GEMM is done separately from the GPU-only case. The tuner for the hybrid CPU/GPU case has two differences from the GPU-only autotuner we have discussed so far. The first difference is that the hybrid tuner extends the search space with one more parameter: the number of parts $p_g$ assigned to the GPU out of $P$. The second difference is that while the single-GPU autotuner optimizes for execution time of the kernel on the GPU, the hybrid tuner optimizes for net system throughput.

There are two assumptions in our current algorithm that allow us to eliminate data copies from the CPU to the GPU on the Ivy Bridge platform. First, we assume that the input matrices are in TN-layout. Second, we assume that the input matrices are aligned to 128-byte boundaries. The alignment restriction allows the creation of an OpenCL buffer from host memory without data copy on Ivy Bridge platform. We expect that such alignment restrictions will not be required on future platforms. For example, the upcoming AMD Kaveri platform will allow completely unified addressing of memory and will not require any data copies irrespective of the alignment.

| Platform | Ivy Bridge | Richland | |
|---|---|---|---|
| Vendor | Intel | AMD | |
| Type | SGEMM | SGEMM | DGEMM |
| CPU-only | 170 | 80 | 40 |
| GPU-only | 140 | 274 | 27.3 |
| CPU+GPU (simple tuned) | 175 | 274 | 57.4 |
| CPU+GPU (fully tuned) | 235 | 274 | 57.4 |

TABLE IV: Hybrid CPU+GPU tuning results (Gflops)

| | Intel Ivy Bridge | AMD Richland | |
|---|---|---|---|
| Type | SGEMM | SGEMM | DGEMM |
| Layout | TN | TN | TN |
| Elements are consecutive | No | No | No |
| Tile size | (8,4,8) | (8,8,2) | (4,8,4) |
| Work-group size | (8,16) | (8,8) | (8,8) |
| SIMD width | 4 | 4 | 2 |
| Bring A to local memory | Yes | No | No |
| Bring B to local memory | Yes | No | No |
| Store A in image | No | No | No |
| Store B in image | No | No | No |
| GPU load (fraction) | 10/16 | 16/16 | 8/16 |

TABLE V: Tuned parameters for GPU kernel and load distribution for hybrid CPU+GPU GEMM

## V. Results for single-chip CPU+GPU systems

### A. Performance

We experimented with Intelś Ivy Bridge platform and AMDś Richland platform (A10-5750M) with Northern Islands GPU. The machine configurations are detailed in Table II. We compared four strategies:

1) CPU-only using vendor-provided CPU BLAS
2) GPU-only using previously obtained kernel tuned for GPU-only scenario
3) CPU+GPU using previously obtained kernel tuned for GPU-only scenario and tuning the load distribution
4) Full CPU+GPU tuner where both GPU kernel and load distribution are tuned together

Performance results are shown in Table IV. On the Ivy Bridge platform, our full tuner outperformed the CPU-only, GPU-only as well as the simple CPU+GPU tuner. We found that the GPU kernel optimized for the GPU-only case was completely different than the kernel optimized for the CPU+GPU case on Intel Ivy Bridge showing that the full co-tuning of GPU kernels and load distribution is indeed required. Optimal GPU kernels for the hybrid case and the load distribution is summarized in Table V.

On the AMD Richland platform, we found that SGEMM was best performed on the GPU alone instead of using both the CPU and GPU. Thus, resource constraints and other overheads outweigh the benefits of a hybrid strategy in this case. This is unsurprising given the imbalanced ratio of CPU/GPU performance on SGEMM on this platform. DGEMM, where the CPU/GPU ratio is more balanced, shows that hybrid strategy outperforms CPU-only or GPU-only solutions.

### B. Power analysis on Ivy Bridge

In single-chip systems such as Intel Ivy Bridge, the chip specifies a maximum power draw. For example, the chip used in this test has a 45W thermal design power (TDP) though it can go slightly above its TDP for short bursts. The sum of the CPU power-draw and the GPU power-draw is not allowed to exceed the specified maximum power draw. In order to maximize performance, platforms such as Ivy Bridge implement dynamic power distribution schemes where the power is distributed between the components depending on the workload. In CPU-heavy workloads, more power is diverted to the CPU while in GPU-heavy workloads more power is diverted to the GPU.

We wanted to understand the relationship of power distribution between components and performance. Intel provides a tool called system analyzer that records power measurement reported by the chip. We used this tool to measure the socket, GPU and CPU power consumption. The socket power consumption is the sum of power consumption of the CPU, the GPU as well as some common parts of the chip such as the memory controller. The measured peak power consumption of each strategy are provided in Table VI. The data confirms that Ivy Bridge has a very dynamic power distribution scheme. Under CPU-only GEMM, the CPU drew up to 36W of power while the GPU was nearly idle. Under GPU-only GEMM, the GPU drew up to 23W of power while the CPU was nearly idle. However, when subjected to the hybrid load, our results show that the chip's power controller decided to throttle both the CPU and the GPU back. Under hybrid load, the power draw of each component is capped by the hardware to about 70-80% of the peak power draw of each component in order to remain under the TDP. Thus, we conclude that in hybrid solutions like Ivy Bridge, the hybrid peak performance cannot be expected to reach the sum of peaks of individual components because under hybrid workloads each component is operating under stronger power constraints compared to when the component is acting alone.

### C. Observations and performance guidelines

We have observed two important considerations when optimizing performance on single-chip systems.

**Explicit flushing of GPU queue***:* Explicitly flushing the GPU queue before the CPU kernel is called is extremely

| CPU load fraction (fraction) | GPU kernel | Performance (Gflops) | Socket power (Watts) | CPU Power (Watts) | GPU Power (Watts) | Performance/(socket power) (Gflops/Watt) |
|---|---|---|---|---|---|---|
| 1 | CPU-only | 170 | 40 | 35.9 | 0.7 | 4.25 |
| 0 | GPU-optimized | 140 | 29 | 2.4 | 22.6 | 4.83 |
| 10/16 | GPU-optimized | 185 | 50.5 | 28.3 | 17.5 | 3.66 |
| 0 | Hybrid-optimized | 110 | 24.3 | 2.3 | 18.6 | 4.52 |
| 10/16 | Hybrid-optimized | 235 | 51.4 | 32.6 | 14.3 | 4.57 |

TABLE VI: Measured peak power for various GPU kernels and CPU/GPU load distributions

important. Without manual flushing, the OpenCL runtime from both Intel and AMD appears to not immediately begin GPU execution and this can reduce performance by as much as 50%.

**Effect of callbacks***:* It is also important to not register any callbacks with the OpenCL runtime. In the initial version of the library, we had registered callbacks with OpenCL to automatically delete GPU buffers when the GPU finishes execution. However, we noticed that upon registering callbacks, the OpenCL runtime was waiting in a loop and consumed significant CPU time. The increased CPU load affected the CPU BLAS performance. The increased CPU load also led to increased CPU power consumption, and the chip often reduced the GPU clocks to remain within limits. Thus, registering callbacks with the OpenCL runtime reduced performance significantly and now we do not register any callbacks with OpenCL.

## VI. Related Work

Autotuning is a well-established technique on CPUs. ATLAS [4] is a well-known example of an autotuning BLAS. Autotuning has also been used for some FFT libraries such as FFTW [5]. However, FFTW uses an online tuner where the tuning happens at application runtime. In contrast, libraries like ATLAS perform offline tuning, where autotuning is performed at install time and hence only done once per machine. Our approach is similar to ATLAS in this regard.

Implementing a fast GEMM routine on GPUs and other accelerators has been of considerable interest in the past few years. Several researchers have written hand-tuned implementations for particular GPU architectures Volkov et al. [6] described a fast GEMM prototype in CUDA for Nvidia's G80 architecture. Their ideas have now been included in Nvidia's CUBLAS library. Nakasato [7] described a fast GEMM prototype for AMD's Cypress GPU (which powered products such as Radeon 5870). Their implementation was written in AMD's CAL API, which was a low-level pseudo-assembler exposed by AMD for their previous-gen GPUs. CAL API has now been deprecated. Nath et al. [8] report a fast CUDA GEMM implementation for Nvidia Fermi GPUs. Tan et al. [9] describe a fast CUDA GEMM implementation for Nvidia's Fermi architecture. They wrote their implementation in PTX, which is a pseudo-assembler for CUDA. This allowed tighter control over instruction scheduling compared to high-level languages like CUDA-C and OpenCL. They report better performance than CUBLAS. Matsumoto et al. [10] reported several high-performance OpenCL GEMM kernels for AMD's Tahiti GPU. Their implementation uses an autotuner to search for parameters, though they limit their experimental evaluation to only one architecture so it is not clear how well it will translate to other architectures. Schneider et al. [11] implemented a fast ZGEMM routine on Cell Broadband Engine that ran on Cellś Synergistic Processing Unit (SPU). They optimized their implementation for the vector instruction set of the Cell processor.

Several researchers have looked at portable OpenCL GEMM implementations for multiple architectures. Du et al. [3] present a study of a portable GEMM implementation. They had two codepaths. One codepath was an OpenCL translation of Fermi GEMM routines from MAGMA's CUDA kernels. This was essentially a handwritten implementation with only a few parameters. The second codepath was an autotuning implementation for AMD GPUs. In comparison, our library offers a unified and completely autotuning implementation for all architectures. In terms of performance, our results show about 15% higher performance on SGEMM on Nvidia GPUs and we get comparable performance on AMD architectures. Weber et al. [12] discussed an autotuning implementation on AMD's GCN GPUs. However, compared to Weber et al., our results show about 60% higher peak performance on Radeon 7970 on SGEMM and about 20% higher performance on DGEMM. Tillet et al. [13] also presented an autotuning framework and showed results for SGEMM and DGEMM on Fermi and GCN architectures. However, our results are almost 40% faster on GCN and 20% faster on Fermi. The reason for our much faster performance is that Tillet et al. are only using one codelet, NN layout with consecutive elements, while our tuner tests 6 different codelets and we show that the TN codelet with offset elements is optimal for these GPUs. Further, their codelet accepted fewer parameters. For example, their codelet does not support copying inputs into OpenCL images.

There is no previous research on tuning GEMM for single-chip CPU+GPU systems. There is also no previous coverage of the Intel Ivy Bridge platform.

## VII. Future work

We have identified directions for future work. The first direction is to develop, or find, an autotuner that generates high-performance CPU code and to use it to extend our hybrid CPU-GPU experiments using such a tuner. In our current setup for hybrid tuning, we tune the GPU kernel and the workload partitioning but use the vendor-provided CPU BLAS on the CPU. Ideally, the CPU kernel used should also be part of the tuning process. However, currently we have not found an satisfactory approach to generating high-performance OpenCL GEMM kernels for CPUs. RaijinCL's autotuner runs on CPUs and in addition we have also developed a prototype block-major layout based generator but we found that our approach currently only reaches about 50-60% of the peak achieved by the vendor BLAS such as MKL. We have observed that this is partly due to immaturity of CPU OpenCL implementations as they often produced some unnecessary register spills. An alternative approach may be to explore a CPU-tuner which

generates multithreaded C code with SSE and AVX intrinsics, instead of OpenCL.

The second direction is to test the performance of our library on more devices such as Intel's Xeon Phi many-core accelerator as well as mobile single-chip systems with OpenCL-capable GPUs such as Qualcomm Snapdragon processors.

Finally, we are also exploring merging our library with other OpenCL libraries such as AMD's clMath library which was recently open-sourced. clMath provides more functions while RaijinCL offers more performance and merging our autotuner with clMath may be an ideal combination of functionality and performance.

## VIII. Conclusions

OpenCL is emerging as a common, portable programming language for a wide variety of compute devices available from many different vendors. However, even though OpenCL is portable, different devices require different OpenCL kernels in order to achieve high performance. In this paper we presented a solution to this problem via the design, implementation and evaluation of RaijinCL, a portable and practical autotuning OpenCL library for GEMM and other matrix operations.

Our autotuning approach was designed by identifying a collection of codelets for each kernel. Different versions of the codelets expose important algorithmic variations. Our GEMM codelets expose the argument layout (transposed or not) and the element assignment scheme (consecutive or offset). Within each codelet we identified important parameters such as tile sizes, SIMD width, work-group size, how to handle local memory and whether or not to use OpenCL images. Given the group of codelets and the parameter space, the autotuner evaluates all points in the search space and identifies the best codelet, and the best parameters for that codelet.

We experimented on a wide variety of GPU architectures from multiple vendors including AMD, Nvidia and Intel. For each device we compared our autotuned library to the vendor's specialized library (when one was available). We found that we sometimes outperformed the vendor's library, and at other times we were close to vendor libraries performance. Thus, we did achieve the goal of having a portable and high-performance solution across a range of GPUs.

In addition to our performance results, we discussed a number of issues about architecture and OpenCL implementations. For example, we provided some analysis of kernels on AMD and Nvidia architectures. We also discussed compiler issues on various platforms such as register allocation and use of FMA instructions that can have significant impact on GEMM performance.

We also extended our autotuner to optimize system-level performance on single-chip CPU/GPU systems where the workload is distributed over both the CPU and GPU. We showed that the GPU kernels optimized for GPU-only GEMM are sometimes different than the GPU kernels optimized for hybrid CPU/GPU GEMM implementation. We found that our autotuning approach successfully generated kernels and load distribution policies that can outperform CPU-only, GPU-only and even naive CPU+GPU tuning policies. We also provided guidelines for carefully managing CPU-GPU synchronization based on our experience in developing RaijinCL.

We provided an overview and analysis of the Intel Ivy Bridge GPU architecture, which has not previously been described in the academic literature. We also presented results on power usage of various CPU and GPU solutions on Ivy Bridge and discussed how power management schemes affect the performance achieved in hybrid workloads compared to CPU-only or GPU-only workloads.

RaijinCL is open source, and we hope that users will use it on many different devices. In addition to performance, we have also paid attention to issues such as deployment and careful API design. As we gain more experience and feedback from users we may be able to further tune the library by exposing further codelets and additional parameters and we hope that our library will be useful for users of many different types of devices.

## References


[1] "The OpenCL Specification," http://www.khronos.org/opencl.

[2] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, Sep. 1979.

[3] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, vol. 38, no. 8, pp. 391 – 407, 2012.

[4] C. W. Antoine, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, p. 2001, 2000.

[5] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[6] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[7] N. Nakasato, "A fast GEMM implementation on the Cypress GPU," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 50–55, Mar. 2011.

[8] R. Nath, S. Tomov, and J. Dongarra, "An improved Magma GEMM for Fermi graphics processing units," *International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 511–515, November 2010.

[9] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of DGEMM on Fermi GPU," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 35:1–35:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063431

[10] K. Matsumoto, N. Nakasato, and S. G. Sedukin, "Implementing a code generator for fast matrix multiplication in OpenCL on the GPU," Graduate School of Computer Science and Enginering, The University of Aizu, Tech. Rep. 2012-002, July 2012. [Online]. Available: ftp://ftp.u-aizu.ac.jp/u-aizu/doc/Tech-Report/2012/2012-002.pdf

[11] T. Schneider, T. Hoefler, S. Wunderlich, T. Mehlan, and W. Rehm, "An optimized ZGEMM implementation for the Cell BE," in *Proceedings of the 9th Workshop on Parallel Systems and Algorithms (PASA)*, Dresden, Germany, Feb. 2008, pp. 113–122.

[12] R. Weber and G. Peterson, "A trip to Tahiti: Approaching a 5 Tflop SGEMM using 3 AMD GPUs," in *Symposium on Application Accelerators in High Performance Computing (SAAHPC), 2012*, 2012, pp. 19–25.

[13] P. Tillet, K. Rupp, S. Selberherr, and C.-T. Lin, "Towards performance-portable, scalable, and convenient linear algebra," in *5th USENIX Workshop on Hot Topics in Parallelism*, 2013.