Adaptive Auto-Tuning of Computations on Heterogeneous Environments

Melissa Castillo and Christian Curley
Dept. of Electrical and Computer Engineering
University of New Mexico
Albuquerque, USA

Carlos Reyes
Technical Mentor
Stellar Science
Albuquerque, USA

Abstract

Heterogeneous computing offers increased performance making it popular for mathematically intensive computations. One of the commonly used programming tools for heterogeneous computing is OpenCL, which stands for Open Computing Language. OpenCL has been proposed as an industry standard framework that allows programs to execute on heterogeneous platforms, from multiple processor vendors. OpenCL, like others frameworks similar to it, suffers from poor performance portability. Kernel programs tuned for a particular processing unit need re-tuning on another device to achieve similar performance. Machine learning based auto-tuning provides one possible solution to the performance portability issue. In this paper, we attempt to demonstrate how the Random Forest machine learning model can be applied to the auto-tuning problem. The naive matrix multiplication algorithm is given certain tuning parameters, and the sample data for varying parameters is used to build and train the Random Forest model. The model can then be used to determine suitable tuning parameters for kernel optimization independent of processor model.

Keywords: auto-tuning, heterogeneous computing, machine-learning, OpenCL, random forests, SGEMM

Introduction

For years central processing units (CPUs) were the only processors capable of performing mathematica--lly intensive computations, but growing demand for faster and cheaper hardware led to the development of accelerated processing units (APUs), and graphics processing units (GPUs). Originally these APUs and especially GPUs, were designed with one purpose and were either incapable or inefficient at performing the numerous functions that CPUs could achieve.

Today the common heterogeneous system is one with a few CPUs cores and thousands of GPUs cores that can perform mathematically intensive computations in parallel. While this parallelism is widely advantageous, implementing programs that can take advantage of the entire system remains challenging.

OpenCL was developed to take advantage of the new capabilities of the modern GPU architecture, and attempt to solve the difficulties of CPU-GPU interoperability. While OpenCL is capable of tying programs to heterogeneous systems, it struggles to do so with constant execution time. OpenCL kernels are programs that can be executed on different devices in a heterogeneous system. These kernels offer portabil--ity, but the performance of the kernel can vary dramatically based on the device it is currently executing on.

One common solution to this issue of performance portability is auto tuning, the concept is to automatically take execution time measurements on

various implementations of the kernel and choose the best implementation for that device. Auto tuning can be empirically driven, model driven, or machine learning driven. Empirically driven auto tuning goes through the entire parameter space to find the best implementation, while it guarantees that the best kernel configuration will be found among the parameter space, it will be a tedious endeavor if the parameter space is large. Model driven auto tuning uses a performance model that is used to find similar kernel candidates to be used. Model driven auto tuning is dependent on the performance model used, if the performance model is inadequate all possible kernel candidates will also share the inadequacy. Machine learning can be used to generate and tune a performance model. By executing a set of random kernel implementations and calculating their execution times, a prediction can be determined for the best kernel implementation.

The entirety of this paper will demonstrate why auto tuning kernels is necessary and how machine learning could be used in auto tuning to generate performance portable OpenCL kernels without the need to exhaustively search the entire kernel parameter space, or manually without creating a performance model.

Related Work

Other, more successful, research has been done related to heterogeneous computations with matrix multiplication [1]. Other auto tuning papers using Artificial Neural Networks have reported solid outcomes [2].

The primary differences between what our paper attempts to accomplish and the research of others is the type of kernel used, the kernel parameters used for tuning, and machine learning techniques employed. In other auto tuning research, the most popular machine learning algorithm employed was Artificial Neural Networks. Random Decision Forests is relative new machine learning model to be adapted to solve the auto tuning problem.

Background

OpenCL

OpenCL is a proposed industry standard framework for heterogeneous computing. It has widely been implemented into the industry with names like AMD, Apple, Intel, Nvidia, Qualcomm, and over 100 other companies using the framework.

It allows programmers to access the OpenCL compatible devices that include CPUs, GPUs, ALUs, and FPGAs. The ability to use the same code across various devices encourages code reuse, but biggest disadvantage to OpenCL and other heterogeneous APIs is performance variations between devices.

Random Decision Forests

The algorithm was first implemented by Tin Kam Ho of AT&T Bell Labs [3]. Later implementations were developed such as Leo Breiman's trademarked Random Forest that uses bagging techniques alongside Ho's original algorithm [4].

Single Precision Floating Point General Matrix Multiplication

Single Precision Floating Point General Matrix Multiplication or SGEMM is any matrix multiplication algorithm that processes single precision floating point elements. Matrix multiplication is a commonly used mathematical operation that has applications across all science and engineering disciplines. Matrix multiplication in its naive form is calculated as:

$$c_{ij} = a_{i1}b_{1j} + \ldots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj}$$
$$where\ i = 1 \ldots n\ and\ j = 1 \ldots p$$

The computational complexity is $O(n^3)$ for the naive matrix multiplication algorithm. Divide and conquer variations of the algorithm that take advantages of parallel processing has allowed for faster execute times.

Implementation

Our benchmark test uses the classic single precision general purpose matrix multiplication (SGEMM). We placed a constraint for the dimensions of the matrices to be square. Input matrices A and B, as well as the output matrix C have the same row and column dimensions.

Benchmark Kernel	Description
Single Precision Floating Point General Matrix Multiplication (SGEMM)	Matrix Multiplication of several sizes of square input matrices. 1x1, 2x2, 4x4, 8x8, 16x16, 32x32, 64x64, 128x128, 256x256, 512x512

Our parameter space consists of the following values:

Parameters	Test Values
MATRIX DIMENSION (GLOBAL WORK SIZE)	1, 2, 4, 8, 16, 32, 64, 128, 256, 512
USE_LOCAL_MEMORY	0, 1
BLOCK SIZE (LOCAL WORK SIZE)	1, 2, 4, 8, 16

Originally the parameter space was designed to include global work group size x, global work group size y, local work group size x, and local work group size y. We had to limited the parameter space down to BLOCK SIZE (local work size) and MATRIX DIMENSION (global work size) because our matrices were specified as square matrices of 2^N x 2^N in the range where $2^N = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$. The parameter space was also limited up to 512 due to the OpenCL constraints of the hardware. OpenCL global work size must the size of the job size, in our case the size of the matrix.

According to the OpenCL specification [5] the number of work groups can be computed as:

$$(W_x,\ W_y) = (G_x / S_x,\ G_y / S_y)$$

where W_x is the number of work groups in the x-dimension, W_y is the number of work groups in the y-dimension, G_x is the index space of the work items in the x-dimension, G_y is the index space of the work items in the y-dimension, S_x is the size of the work group in the x-dimension, S_y is the size of the work group in the y-dimension.

We build our kernel using the OpenCL API for C. The C program is the OpenCL host code that sets up the execution environment for the kernel on a particular device. The host code contains functions to establish an OpenCL context, OpenCL command queue, OpenCL program, and OpenCL kernel. The kernel will operate on a work environment that is defined by clEnqueueNDRangeKernel. This function takes the global and local work sizes as parameters and these parameters depend on the size of the matrix, as well as the maximum allowed work size for the device.

The host function contains an OpenCL function for event profiling that determines that kernel execution start and finish times. The difference between the two determines the execution time of the kernel in nanoseconds. For convenience, we shifted the results from nanoseconds to milliseconds. The kernel parameters and their corresponding execution times are then emitted to a CSV (comma separated values) file that will be parsed by a Python3 script. This Python script contains the Random Forest algorithm, which was taken from a popular module called Scikit Learn.

We choose Random Forest as our machine learning algorithm because our nature of Random Forest fits the problem well. Our parameter space consists of features that have varying ranges from 1 to 512, 0 to 1, and 1 to 16. Random Forest is a nonlinear machine learning algorithm and responses well with features that have variable ranges. The Random Forest algorithm also doesn't need to be normalized between 0 and 1, in fact normalization would smooth out the nonlinearity of the model. If normalization were to occur with our parameters, it would be difficult to denormalize back to the original values. Normalization would make sense if all the parameters operated within the same range of values.

Once the set of kernel implementations is feed into the random forest regressor, a prediction model is generated and undergoes training to obtain accurate predictions for kernel parameters that have not been provided in the original dataset.

The Python script then generates predictions from the kernel parameters of the test set and with the actual execution times. The error is generated between the predicted and actual. Then the best actual kernel implementation is found.

Testing was performed on the following devices:

Test Hardware / OS	OpenCL Device Specs
Intel i5-3317U / macOS High Sierra 10.13.4	Max Compute Units: 4 Max Work Group Size: 1024 Max Work Item Dimension 3 Max Work Item Sizes: 1024 / 1 / 1 Max Clock Frequency: 1700 MHz

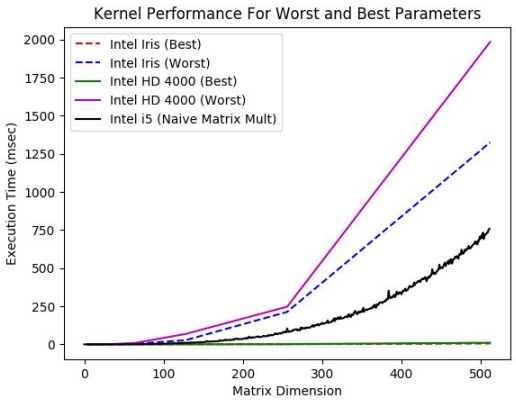
Intel i7-4578U / macOS High Sierra 10.13.4	Max Compute Units: 4 Max Work Group Size: 1024 Max Work Item Dimension: 3 Max Work Item Sizes: 1024 / 1 / 1 Max Clock Frequency: 3000 MHz
Intel HD Graphics 4000 / macOS High Sierra 10.13.4	Max Compute Units: 16 Max Work Group Size: 512 Max Work Item Dimension: 3 Max Work Item Sizes: 512 / 512 / 512 Max Clock Frequency: 1050 MHz
Intel Iris 1536 MB / macOS High Sierra 10.13.4	Max Compute Units: 40 Max Work Group Size: 512 Max Work Item Dimension: 3 Max Work Item Sizes: 512 / 512 / 512 Max Clock Frequency: 1200 MHz

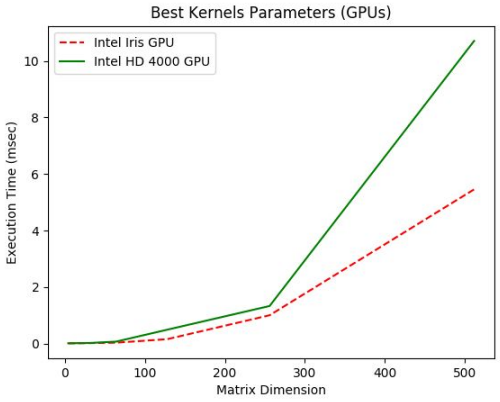
Due to resource constraints, the amount of available hardware to test and compare kernel performance was limited to the devices shown in Table 3.

Results

We performed various tests on our method by using the matrix multiplication kernel for varying dimensions for the input matrices, see Table 1 for the list of different input dimensions.

The OpenCL kernel was sampled for the different matrix multiplication dimensions on the Intel Iris GPU, and the Intel HD Graphics 4000. The Intel i5-3317U CPU was running the naive matrix multiplication with no OpenCL to compare the performance of the kernels to the naive implementation. The results below illustrate that the worst kernel parameters can execute 2x slower than a naive matrix multiplication on a CPU.

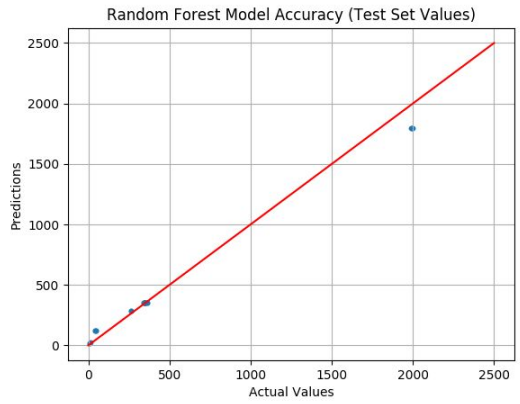




The best kernel parameters on the GPUs show that for matrices of 512 x 512, the execution times are 10 milliseconds or less. The Intel Iris GPU exhibits faster, about 2x faster, execution times than the Intel HD Graphics 4000. This is most likely due to the fact that the Intel Iris has 2.5x more compute units than the Intel HD Graphics 4000, and the Iris also has a max clock frequency that is 14.28% faster. This combination of more compute units and high clock frequency means that Iris would theoretically have greater parallelism and execute more instructions faster than the HD Graphics 4000 counterpart.

Unfortunately, our dataset was extremely small due to the fact that only 3 kernel parameters were used with values of limited range. Going through every possible iteration of the parameter space results in a dataset of 66 possible samples, and some of these kernel parameters might not execute because local work size cannot be larger than the global work size. Machine learning algorithms are built with large datasets in mind, on the order of millions of samples. This means that it extremely difficult to generate accurate models for small datasets. The alternative method to find the best kernels for small datasets was

to perform an empirical auto tune, though an exhaustive search and plot the results into a separate Python script to illustrate the kernel comparisons.



The Random Forest machine learning prediction model was executed 100 times to obtain an average prediction accuracy of 95%, but because the dataset was limited we believe these values are due to overfitting of the Random Forest model. At this time, we cannot guarantee that the model prediction accuracy is truly 95% due to the highly likelihood of overfitting.

Conclusions and Future Work

This paper set out the prove that adaptive auto tuning is necessary in heterogeneous computing and that Random Forest machine learning could be a useful tool for auto tuning. Our results demonstrate that the way the OpenCL host code sets up the kernel for execution can dramatically affect the performance of the kernel. If the kernel parameters are set incorrectly, the execution time can perform 2x worst than the naive matrix multiplication running on a CPU. On the positive side, if you were to tune the kernels to quality parameters, the kernels obtain an estimated 75x speedup factor over the same naive matrix multiplication.

When it comes to the machine learning aspect of this project, it is hard to say if Random Forest machine learning can effectively be used in auto tuning. The only way to determine effectiveness of Random Forest is to generate larger datasets to remove the overfitting. Future work could be done to increase the amount of samples in the dataset. The best way to do this would to increase the number of features of the Random Forest. Besides global work size, local work size, local memory, and global memory, additional kernel parameters could be added, like OpenCL texture memory usage, and loop unroll factors for matrix multiplication. Also, opening the matrix dimensions to different sizes besides 2^N could open more options for the global work size and local work size. We also need to keep in mind that some additional kernel parameters, such as loop unrolling, are features unique to the matrix multiplication kernel. Developing other kernels besides matrix multiplication would open up new features unique to that problem.

The execution time of the program could be dramatically improved in two areas. First, the comparison check to see if the kernel returns the correct result could be improved. The check is needed to ensure that kernels not only run fast but return the right result. There is likely a more efficient method to perform kernel error checking. Second, the Random Forest prediction generation could also be improved. For our purposes, Python and the SciKit Learn module was sufficient enough to generate a prediction model, but if this were to be used in real time applications the program would be extremely slow. Possibly the best solution to improve the prediction generation would be to implement the Random Forest algorithm using C/C++ instead of Python.

The OpenCL specification is being restructured into a new API called Vulkan. Vulkan combines the heterogeneous capabilities of OpenCL with the graphics capabilities of OpenGL. This decision to restructure OpenCL to Vulkan means that our program has a high risk of containing deprecated functions with one to two years from the date of this paper. Any future work direct related to this project will have to restructure the code to meet the new Vulkan specification.

Acknowledgments

The authors would like to thank Carlos Reyes, the technical mentor and sponsor for this project. He provided a wonderful and challenging project to address. This was a project that none of us had prior knowledge about, and we like to thank Carlos for the new knowledge we gained by tackling this problem. The authors would also extend thanks to the wonderful employees at Stellar Science for allowing project discussions to take place in their office.

References

[1] Matsumoto, Kazuya, Naohito Nakasato, and Stanislav G. Sedukhin. "Performance tuning of matrix multiplication in OpenCL on different GPUs and CPUs." *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012.

[2] Falch, Thomas L., and Anne C. Elster. "Machine learning based auto-tuning for enhanced opencl performance portability." *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015.

[3] Ho, Tin Kam. "The random subspace method for constructing decision forests." *IEEE transactions on pattern analysis and machine intelligence* 20.8 (1998): 832-844.

[4] Breiman, Leo. "Random forests." *Machine learning* 45.1 (2001): 5-32.

[5] Khronos OpenCL Working Group. "The OpenCL specification version 1.2." *https://www. khronos. org/registry/OpenCL/specs/opencl-1.2. pdf* (2012).

[6] Scarpino, Matthew. "OpenCL in action." *Westampton: Manning Publications* (2011).

[7] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html