GPU CUDA Kernel Detection from Graphical Algorithm Representations

Frank Zhao, Australian National University
Supervised by Dr. Eric McCreath, Australian National University
frank.zhao@anu.edu.au

January 23, 2015

Abstract: An experimental framework for generating CUDA kernels using graphical representations of algorithms is presented. Kernels and threads are identified using the disconnectivity and isomorphic nature of these graphs. It is shown that kernels generated using this method exhibit pre-alignment of input memory. For square matrix multiplication, a GPU kernel speedup of up to 2.8× compared to a handwritten naive implementation was observed.

1 Introduction

Graphical processing units (GPUs) are now well known for their capability for general purpose parallel computing. The high number of cores and large memory bandwidth found in many GPUs provide a great platform for highly parallel computing. However, parallelising serial code for execution on a GPU and doing so in an optimal way still remains an active research topic.

Parallelising existing sequential code for execution on GPUs remains an interesting problem. There exist solutions such as OpenMP [1] and OpenACC [2] which utilise pragma statements to unroll loops and generate parallelised kernels for execution. The program implemented for this paper (Codegraph) experiments with an additional abstraction of serial code parallelisation by analysing graph structures that are produced by tracing operations performed on data during code execution.

One of the largest bottlenecks in optimising code for execution on parallel GPU systems is the bottleneck encountered in memory copying and access. These problems are especially evident in naive implementations, where little care is given to optimisations during runtime, such as memory alignment. Nowadays it is well known that having an optimised memory mapping provides a significant boost in performance on GPU kernels of up to 3.3× [3].

1.1 Algorithms as Graphs

In its simplest definition, and algorithm is a sequence of operations performed on a collection of input values that result in a transformed collection of output values. If we consider manipulated values in the process of executing the algorithm as nodes, and the operations on these values as edges, we can form a graphical representation of the execution trace of the algorithm on a collection of input values. These graphs contain all the information required to create the collection of output values from the inputs - nodes without incoming edges form the initial values and similarly, nodes without outgoing edges can be identified as the resulting values after the algorithm is executed. The paths from the outgoing edges can be traced back to a collection of incoming nodes, this provides both an indicator of dependencies as well as a sequence of operations needed to transform from the initial values to the output.

Analysing these graphs, it is immediately obvious that each disconnected subgraph can be identified as an individual execution thread. Upon further analysis, it is evident that isomorphisms exists between sets of disconnected subgraphs. Each set of isomorphic subgraphs can then be identified as the unique kernel that is executed on the containing subgraphs, or threads.

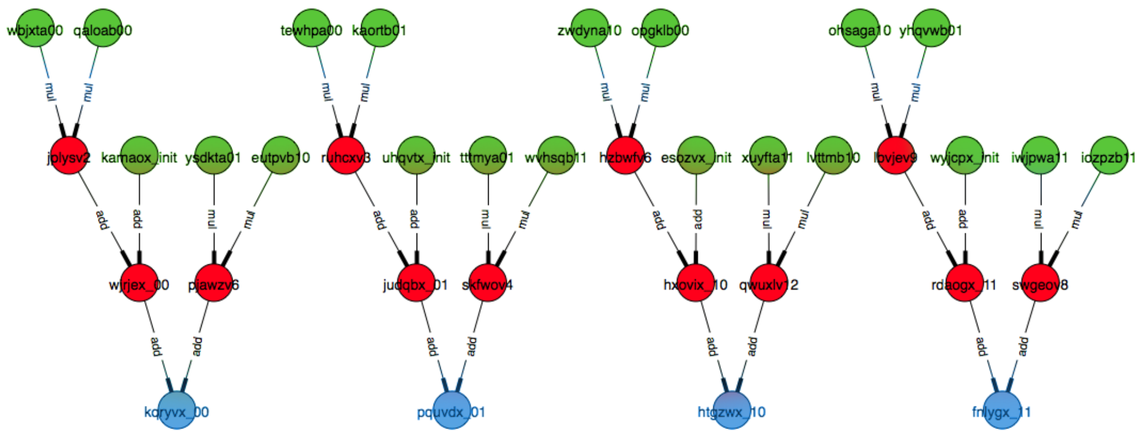

FIGURE 1. An example graph for 2×2 square matrix multiplication. Green indicate nodes with no incoming edges (initial values), red indicates nodes with no outgoing edges (output values). Operations between nodes are represented and labeled as edges.

Since the graph contains all the information needed to reproduce the effects of the algorithm, the code for each individual kernel can be generated. Likewise, the input data for each thread can aggregated into contiguous chunks, all of which form the initial memory that needs to be allocated and copied to the GPU.

2 Matrix Multiplication

The particular example looked at in this experiment was matrix multiplication. Matrix multiplication is an inherently parallelisable algorithm, since computation can be divided into independent computations. A simple naive implementation would assign a thread for computing a multiplication and addition for each row-column pair.

In this simple example, the graph edges consist of addition and multiplication operations, while the nodes consists of values in the initial matrices, the values in the output matrix, and any intermediate values. A example of a generated graph for 2×2 square matrix multiplication is shown in Fig. 1.

3 Codegraph

Codegraph is a pre-compilation optimisation tool. Codegraph takes an input of nodes and directed connecting edges, and identifies the unique kernels necessary to reproduce the output values. Disconnected graphs, or threads, are identified using a flood fill. Disconnected graphs are then arranged in isomorphic groups. Each unique group can then be identified as a CUDA kernel.

In order to generate the code inside the kernel, output values are traced to the initial nodes. The path traversed then forms the sequence of operations needed to reproduce the final value from then input nodes. Since each disconnected graph forms a single execution thread, the input values can be grouped into 'chunks' on a per thread basis, such that required memory regions are aligned. This achieves contiguous memory access for each thread. These chunks are then combined to form one block of memory that maximises cache hits for neighbouring threads.

Codegraph is currently only designed for algorithms with only numerical operations that can be represented as a sequence of additions and multiplications. While this only represents a small subset of computational algorithms, the natural simplicity of graph structures allows for other operations to be implemented easily.

4 Results

Results were based on execution on three NVIDIA GTX 580 GPUs. Each GPU has 512

CUDA cores at 1.56 GHz with 1.5 GB of device memory and 48 KB of shared memory per block. CUDA driver 6.0 was used with a compute capability of 2.0. GPU kernel execution times were measured from the kernel launch to the kernel exit. Overall performance was measured from the beginning of the memcpy from host to device to after the output data is copied from device to host. The host machine consists of a Intel Core i7 960 CPU at 3.20 GHz with 12 GB of memory.

Measuring performance on square matrix multiplication, executing kernels generated by Codegraph resulted in a GPU kernel speedup of up to 2.8× compared to handwritten naive matrix multiplication CUDA kernel implementation. These results are comparable to memory optimisation techniques conducted by Che et. al. [3]. Because of this, the performance increase seems to be primarily attributed to the contiguous grouping exhibited in the device memory.

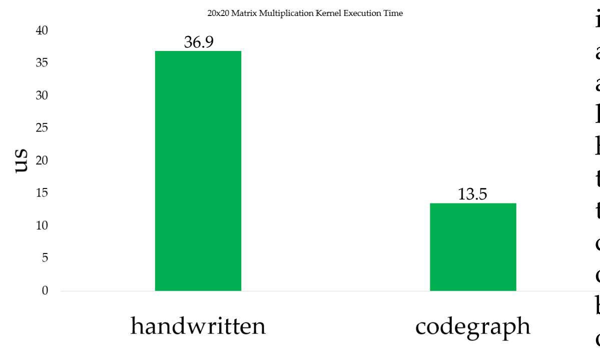

FIGURE 2. Performance of Codegraph generated code compared to a naive implementation. The algorithm in this case was for 20×20 square matrix multiplication.

5 Discussion

Since the memory chunk for each disconnected graph contains all the initial values, there exists some duplication between chunks. In the example of matrix multiplication, the initial values in the output matrix are included in every chunk. Despite this overhead, a small overall performance gain was still observed, of around 1%. Although this redundant copying can be removed manually after preprocessing, performing this automatically during the preprocessing stage would be an interesting task to investigate.

Another possible extension for the preprocessing stage would be to implement automated graph structure generation using directives in sequential code. Currently graphs are generated by inserting node and edge object creation statements into the serial code. A useful extension to this would be to implement OpenACC-like pragma statements that automatically unroll loops and generate kernels for parallel execution.

Some studies have experimented with on-the-fly optimisations [4]. Perhaps these frameworks can find mutual benefit in considering graph based techniques to identify possible optimisation techniques. By considering graphical representations of traditionally serial algorithms, the parallel optimisation problem can find common ground with more abstract graph analysis techniques. There already exist frameworks for large scale graph processing [5], in addition to the wealth of literature available on graph analysis. Likewise, leveraging the optimisation techniques presented in literature, the Codegraph approach could perhaps achieve even higher memory optimisation. Additional analysis techniques such as those of Zhang et. al. [6] during preprocessing can perhaps achieve automatic determination of optimal kernel launch parameters such as block and thread optimisations based on memory chunking.

Finally, the experimental framework Codegraph is not currently designed for performance. Regardless of this fact, preprocessing code for optimisation can only be considered useful if the overhead of preprocessing is outweighed by resulting performance benefits. While Codegraph seems to show improvements to simple, naive implementations, more work is needed before the graphical optimisation approach can be applied to larger, more complex code. A major factor in this is that the complexity of graph isomorphism remains unknown. However, this does suggest that improving algorithms used in the kernel identification process may prove to be a key step to increasing the performance of the preprocessing.

6 Conclusions

An framework was developed to investigate kernel detection from graphical representation of algorithms. Results indicate that kernel generation from graphical algorithm representations can produce kernels that exhibit pre-aligned memory. Kernels generated with the developed Codegraph framework exhibits such a contiguous memory mapping. Execution of Codegraph generated kernels for square matrix multiplication produced a performance speedup of up to 2.8× when compared to handwritten naive implementations of the same algorithm. Code generation from graphs presents an interesting abstraction to code parallelisation and has potential to be investigated further.

7 Final words

Many thanks to Dr. Eric McCreath for his advice and supervision. Thanks also go out to Andrew Haigh and Gaurav Mitra for their guidance and fruitful discussions. The code for this implementation has been made available on GitHub: `https://github.com/frankzhao/codegraph`.

References

[1] `http://openmp.org/`.

[2] `http://www.openacc-standard.org`.

[3] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011*, 2011. DOI: 10.1145/2063384.2063401, SC 2011, Seattle, WA, USA, November 12-18, 2011.

[4] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *ASPLOS'11, March 5-11, 2011*, 2011.

[5] `http://uzh.github.io/signal-collect/`. Signal/collect is a framework for large-scale graph processing.

[6] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. Streamlining gpu applications on the fly. In *ASPLOS'11, March 5-11, 2011*, 2011.