SCHOOL OF COMPUTER SCIENCE
HIGH PERFORMANCE ARCHITECTURE (COMPSCI 243)

GPU Benchmarking Report

Zahra Montazeri
Nitin Agarwal
May 27, 2016

Contents

1 Vision and Scope

In this project, a set of well-defined test cases for the GPU have been established. This test is used to compare the performance across platforms with different graphics cards. The main focus is to benchmark some parameters of same test cases on different graphic cards. For comparison we did benchmark on three different NVIDIA cards and profile some parameters using nvprof as profiling tool. We have the results as well as analysis on the results at the end of this documentation.
First we searched for GPU programs and performed them on Linux. All benchmarks have been performed using the release 3.1 version of the Rodinia. Later in this report we have the more detail description of these programs.
We used nvporf as NVIDIA profiling tool which is a command-line profiler available for Linux, Windows, and OS X. It helps user to understand and optimize the performance of the CUDA application and easily compare and analyze the result of one benchmark on different GPUs. CUDA 7.5 is the version which we used for all three graphic cards in this project.
There are some parameters in which we compared and analyzed within these three different graphic cards. Calculating execution time is the first parameter we experimented and compare how different NVIDIA cards run same programs. We also consider number of kernel calls and API calls in each program and compare this factor with different machine as well. The percentage of heat released by the program and using fan is another two factors we have covered in this benchmarking. We can also see on which GPU each kernel ran, as well as the grid dimensions used for each launch. Gridsize is Number of blocks in a grid along the X and Y dimensions for a kernel launch. This is very useful when we want to verify that a multi-GPU application is running as we expect. But in this profiling the applications are running just on one GPU. The last but not least parameter we considered for comparing and analysis is regs of each program which is number of registers used per thread for a kernel launch.

2 Methodology

2.1 GPU Parallelisation

The GPU computing approach uses graphic card to perform the calculation in parallel. This approach is based on CUDA which is made by NVIDIA and works on NVIDIA graphic cards that have Compute Capability 2.0 or higher. The point is worth mentioning that only hydrodynamic calculations are performed on the GPU and the additional calculations are ran on the CPU. Then these calculations are parallelised using shared memory approach, OpenMP.

2.1.1 Hardware

Following are the graphic cards which we used for performing benchmarks:

GPU	Compute Capability	Number of CUDA cores	Memory (GB)
GeForce GTX 530	XXX	XXX	XXX
Geforce GTX 550	XXX	XXX	XXX
Quadro 6000	XXX	XXX	XXX

GPU specifications

following hardware platforms have been used:

	Compute Model	Processor	Memory (GB)	Operating system
1	DELL Precision T5400	XXX	XXX	XXX
2	XXX	XXX	XXX	XXX
3	XXX	XXX	XXX	XXX

Hardware platforms

2.1.2 Software

All benchmarks have been performed using the release 3.1 version of the Rodinia: accelerating Compute-Intensive Applications with Accelerators. It released to address some concerns in which platforms face. For example, there are many suites for parallel computing on general-purpose CPU architectures, but accelerators fall into a gap that is not covered by previous benchmark development.

3 Description of Benchmarks

Here are brief description for all of the benchmarks we used in this project.

3.1 Breadth First Search

Graph algorithms are fundamental and widely used in many disciplines and application areas. Large graphs involving millions of vertices are common in scientific and engineering applications. This benchmark suite provides the GPU implementations of breadth-first search (BFS) algorithm which traverses all the connected components in a graph.

3.2 K-Nearest Neighbor

NN (Nearest Neighbor) finds the k-nearest neighbors from an unstructured data set. The sequential NN algorithm reads in one record at a time, calculates the Euclidean distance from the target latitude and longitude, and evaluates the k nearest neighbors. The parallel versions read in many records at a time, execute the distance calculation on multiple threads, and the master thread updates the list of nearest neighbors.

3.3 B+ Tree

B+ Tree application has many internal commands that maintain database and process querries. Only J and K commands had enough parallelism to be ported to parallel languages (OpenMP, CUDA, OpenCL). In these implementations, in case of both J and K, the same algorithms (optimized for exposing fine-grained parallelism) were used for fair comparison purposes. For C/OpenMP execution, it is possible to use the original algorithm.

3.4 PathFinder

PathFinder uses dynamic programming to find a path on a 2-D grid from the bottom row to the top row with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally ahead. It iterates row by row, each node picks a neighboring node in the previous row that has the smallest accu- mulated weight, and adds its own weight to the sum. This kernel uses the technique of ghost zone optimization.

3.5 LU Decomposition

LU Decomposition is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix.

3.6 Gaussian Elimination

Gaussian Elimination computes result row by row, solving for all of the variables in a linear system. The algorithm must synchronize between iterations, but the values calculated in each iteration can be computed in parallel.

3.7 SRAD

SRAD (Speckle Reducing Anisotropic Diffusion) is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs). It is used to remove locally correlated noise, known as speckles, without destroying important image features. SRAD consists of several pieces of work: image extraction, continuous iterations over the image (preparation, reduction, statistics, computation 1 and computation 2) and image compression. The sequential dependency between all of these stages requires synchronization after each stage (because each stage operates on the entire image). SRAD is also uses as one of the initial stages in the Heart Wall application. Partitioning of the working set between caches and avoiding of cache trashing contribute to the performance. In CUDA version, each stage is a separate kernel (due to synchronization requirements) that operates on data already residing in GPU memory. The code features efficient GPU reduction of sums. In order to improve GPU performance data was transferred to GPU at the beginning of the code and then transferred back to CPU after all of the computation stages were completed in GPU. Some of the kernels use GPU shared memory for additional improvement in performance. Speedup achievable with CUDA version depends on the image size (up to the point where GPU saturates).

4 Benchmarking using a GeForce 530

These tests have been performed using a GeForce 530 and hardware platform 1 specified in Table XXX.

5 Benchmarking using a GeForce 550

These tests have been performed using a GeForce 530 and hardware platform 1 specified in Table XXX.

6 Benchmarking using a Quadro 6000

These tests have been performed using a GeForce 530 and hardware platform 1 specified in Table XXX.

/sectionResults

/sectionAnalysis

/sectionConclusion

/sectionReferences