Bitmap Manipulation with CUDA and Cache Optimization

12/14/2015

Kyle Berry

William Weber

CPEG 455

Design:

Our code is divided into two optimizations that perform the same task, bitmap manipulation. One optimization utilizes cache optimization features (loop unrolling), while the other is implemented in CUDA so as to be run on a GPU.

Each version is run multiple times to measure performance and consistency. The results of these programs (the modified bitmap images) are verified independently once they've completed to ensure the operations on both bitmaps performed correctly.

Performance Results:

CUDA - 0.067 Seconds (avg, at 1024x1024)

0.0002 Seconds (avg, at 1024x1024 dimension)(CUDA-ONLY operation)

CUDA - 0.0044 Seconds (avg, at 256x256 dimension)

0.000017 Seconds (avg, at 256x256 dimension)(CUDA-ONLY operation)

CPU - 0.0466 Seconds (avg, at 1024x1024 dimension)

CPU - 0.0030 Seconds (avg at 256x256 dimension)

CPU(base-line) - 0.051 seconds (avg, at 1024x1024 dimension)

CPU(base-line) - 0.0032 seconds (avg, at 256x256 dimension)

Analysis:

We determined early on in the project that the most effective spot to speed up the process would be in one of two spots. The first, being in the image creation stage where coloration is applied to the pixels after the code necessary to build the header. It can be expanded to a relatively simple for loop that would simply color the remaining bytes in the image strip. Second, the process of actually writing the array of characters to the file descriptor seemed to be the second most linear task, and thus the next optimal candidate for optimization. In order to achieve a decent speedup we found the loop unrolling was able to reduce the total instruction count by reducing the number of branch statements. This yielded a 10% speedup from the baseline-not unrolled implementation, with that percentage increases in speed remaining consistent across varying resolutions. We were able to optimize both chokepoints that we determined, each independently adding to our speedup.

Second, we attempted to port the operations to the GPU, which proved to be a much more difficult task. After compensating for the offset of the bitmap format, and attempting to parallelize the character array and file descriptor writing, we found that CUDA does not support the capability of writing directly to a file descriptor. Particularly, it is not able to execute the putc function. This set us back for a while after trying to find work arounds, but we eventually settled on only parallelizing the coloration function. Since it was the only operation that we were offloading to the GPU it was relatively simple to see that the time spent on the GPU operation was relatively small, however the overall speed-down occurred because of the copying that was conducted from the host to the device and back to the host, which is not an issue when optimizing a task for cache-centric operations.

So, in conclusion, we determined that a cache-based optimization has been the overall better solution for the given problem of generating bitmap images by comparing the architecture features of two possible ways you could go about implementing the same solution.

Architecture Features

NVidia Geforce GT 620 - Using Cuda 7.5

Intel Xeon E5530 @ 2.4GHz - Using GCC 5.2.0

How to Run:

We’ve included a Makefile with the rest of the course project and it only depends on needing the Nvidia CUDA compiler and gcc. In order to build the project, one needs only to issue a ‘make simple’ to build the naive implementation that creates the ‘simple’ executable file. In order to build the rest of the project, the requisite make commands are ‘make cache’ and ‘make gpu’ that should output the executables for the corresponding tests. All of the above executables have been tested in an Arch Linux environment, but have also been tested on ubuntu 14 for compatibility.

In order to change the resolutions to those noted in the results sections, one most modify the N variable at the top of the bmp.cu file and the bmpCache.c file. For testing the cuda version, you must also change the block dimensions if you are changing the image resolution. For 1,024, you must keep the block dimensions at 32. For 256 elements you must change the block dimensions to 16.