## Problem Statement

The goal of this project is to learn the basics of OpenACC and apply them in the Bridges computing environment. The underlying problem is to solve a Laplace equation for the steady state heat exchange on a two dimensional plate. The algorithm works in iterations, where each iteration averages the points surrounding each point on the plate. This average is then applied to each respective point at the end of the current iteration. Once the change in value between iterations has reaches an acceptable threshold, the algorithm is complete and we have the steady state value for each point on the plate. This simple problem speeds up nicely in parallel environment and gives us an easy algorithm to learn the basics of OpenACC and parallel computing.

## Approach to Solution

The source code is written in c and I used a few other software tools to accumulate the data. I used a Makefile for compiling the code with different compilers and on different machines. To aggregate the data I used a bash script to execute the code under different conditions, and collect the data in comma separated text files. I then used R to calculate relevant statistics on the data and produce plots to visualize the data. To save and version my work, I used a git repository on Github. This also proved useful in transferring my data and code to and from the Bridges machine to my local machine.

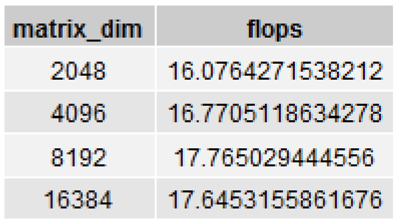
## Solution Description

While the code for this assignment is almost identical to the code from the previous assignment, the implementation is totally different. Assignment one was executed on traditional computer processors running in parallel. This assignment utilizes the NVIDIA K80 and P100 graphics processing units. These GPUs have thousands of processor cores that can all operate in parallel. Unfortunately we don’t have the ability to scale the number of processors. Bridges only allows us to request all or none of the GPUs. Therefore we will not be able to get the granularity of data that we obtained in the first assignment. To calculate the number of floating point operations per second (FLOPS), we use the same formula as the first assignment given in formula (a). Further graphics will represent the performance in Giga FLOPS.

(a) $$FLOPS = Iterations * Matix\ Dimension^{\wedge}2 * 5 \ / \ Execution\ Time$$

We knew from the start that copying the data back and forth between the CPU and the GPU was going to consume a large amount of time. To gain a better perspective of the parallelization potential of the problem, we want to do everything we can to minimize this overhead. Using the compile time output in Appendix A and the run time code profiling in Appendix B, we can see that only about 2.75% of the total execution time is spent copying the memory in and out of the

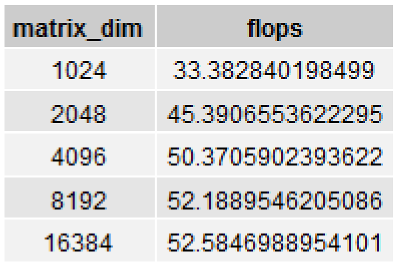
GPU. Figure 1 shows the performance on the K80 GPUs as the memory size increases. We can see that the overhead becomes amortized across the computation as the problem size grows.



| matrix_dim | flops |
|---|---|
| 2048 | 16.0764271538212 |
| 4096 | 16.7705118634278 |
| 8192 | 17.765029444556 |
| 16384 | 17.6453155861676 |

Figure 1: K80 performance

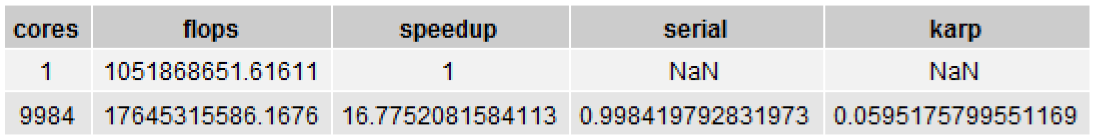
The effect is more pronounced in the P100 and the overall performance is much greater as seen in Figure 2. While the K80 GPUs operate with much more cores, the superior memory bandwidth of the P100 GPUs give it significant performance speedup. The maximum speedup observed was 52.72 GFLOPS using the P100 GPUs. Unlike the performance data in the first assignment, the data was much more uniform and the standard deviation was much smaller.



| matrix_dim | flops |
|---|---|
| 1024 | 33.382840198499 |
| 2048 | 45.3906553622295 |
| 4096 | 50.3705902393622 |
| 8192 | 52.1889546205086 |
| 16384 | 52.5846988954101 |

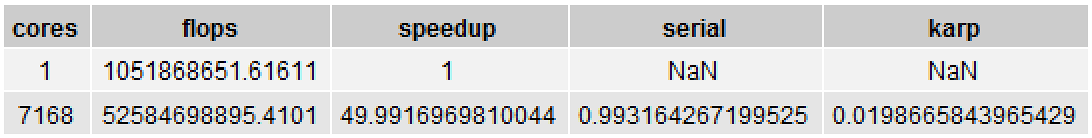
Figure 2: P100 performance

Since we can't scale the number of GPU cores that are utilized, we have to compare the execution on the two different GPUs with the serial execution on the CPU. This is less than ideal because they are completely different architectures and don't really follow the parameters for the Karp-Flatt and Gustafson-Barsis calculations. We can see in figure 3 that the serial execution calculations from show that over 99% of the execution is serial code. We know this to be false based on the data we have from Appendix A and B. It's impossible with our current setup to get accurate data for these calculations. We also are un-able to determine any kind of slope or trend of the Karp-Flatt calculations.



| cores | flops | speedup | serial | karp |
|---|---|---|---|---|
| 1 | 1051868651.61611 | 1 | NaN | NaN |
| 9984 | 17645315586.1676 | 16.7752081584113 | 0.998419792831973 | 0.0595175799551169 |

Figure 3: K80 calculations



| cores | flops | speedup | serial | karp |
|---|---|---|---|---|
| 1 | 1051868651.61611 | 1 | NaN | NaN |
| 7168 | 52584698895.4101 | 49.9916969810044 | 0.993164267199525 | 0.0198665843965429 |

Figure 4: P100 calculations

## Appendix A

Compile time OpenACC operations.

```
[gillespi@gpu002 laplace_acc_debug]$ make
pgcc -acc -O4 -ta=tesla,8.0 -Munroll -mcmodel=medium -Minfo=accel  laplace.c -o
laplace_debug.out
main:
   60, Generating create(Temperature[:][:])
       Generating copy(Temperature_last[:][:])
   65, Loop is parallelizable
   66, Loop is parallelizable
       Accelerator kernel generated
       Generating Tesla code
       65, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
       66, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
   76, Loop is parallelizable
   77, Loop is parallelizable
       Accelerator kernel generated
       Generating Tesla code
       76, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
       77, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
       78, Generating implicit reduction(max:dt)
```

## Appendix B

Run time OpenACC code profiling.

```
[gillespi@gpu002 laplace_acc_debug]$ ./laplace_debug.out
3488, 2048, 4.631465, 15793849626.414103, 4

Accelerator Kernel Timing data
/home/gillespi/Workspace/laplace_acc_debug/laplace.c
  main  NVIDIA  devicenum=0
    time(us): 4,083,049
    60: data region reached 2 times
        60: data copyin transfers: 3
             device time(us): total=5,463 max=2,711 min=46 avg=1,821
        92: data copyout transfers: 3
             device time(us): total=5,455 max=2,869 min=57 avg=1,818
    64: compute region reached 3488 times
        66: kernel launched 3488 times
            grid: [64x512]  block: [32x4]
             device time(us): total=1,630,390 max=476 min=465 avg=467
            elapsed time(us): total=1,747,887 max=747 min=491 avg=501
    75: compute region reached 3488 times
        75: data copyin transfers: 3488
             device time(us): total=25,603 max=303 min=3 avg=7
        77: kernel launched 3488 times
            grid: [64x512]  block: [32x4]
             device time(us): total=2,109,908 max=617 min=603 avg=604
            elapsed time(us): total=2,218,808 max=1,728 min=626 avg=636
        77: reduction kernel launched 3488 times
            grid: [1]  block: [256]
             device time(us): total=230,281 max=76 min=65 avg=66
            elapsed time(us): total=332,169 max=561 min=86 avg=95
        77: data copyout transfers: 3488
             device time(us): total=75,949 max=369 min=14 avg=21
```