Nolan Anderson | Programming Assignment 1 | CPE 613 | Janruary 21 2023

1. Using the code snippets found here, in the lecture slides, and in the provided articles provide
   - **An implementation of the saxpy shown in the slides (**Provided)
   - **A CUDA events based timer using the interface provided and a test to validate that it works** (Provided)
   - **A driver comparing your saxpy implementation to the BLAS, timing of your kernel (averaged over 1000 runs), and achieved computational and memory throughput.**
     - I made a couple changes in test.cpp to include timing for both the fortran and saxpy calls. See Appendix 1 for the code.
   - **A Makefile to build these things (**Provided)

2. **Write a brief report documenting your successes, failures, and performance achieved by running on the Ampere cards on the DMC. Compare with the theoretical maximums in the whitepapers, attempting to deduce what the performance limitation was.**

- Successes / Failures

I believe I succeeded in comparing execution times between the saxpy and fortran calls (see the performance section). The addition of the loop and a couple of timer calls, I was able to do comparison between the saxpy and fortran implementations. I changed the code by adding in a for loop before and after the saxpy/fortran implementations, and calculating the time for execution of both. This required me to divide the total time by 1000 since we are doing 1000 runs. You can see my changes to test.cpp in the appendix below, and I have also copied over my directory to /student_files/anderson for you to run.
For failures, I believe that I should have started this assignment much sooner. I genuinely do appreciate you helping us to understand the assignment, I was a little confused at first. The videos were very helpful! Thank you for providing the code as well. I will get started much sooner next time. This stuff is really cool, and I want to learn it.

- Performance of Ampere cards on the DMC

```
==================
Starting execution
==================

        - Fortran Execution Time:   0.002523 Seconds
        - Saxpy Execution Time:     0.000094 Seconds
        - Computational Rate:       2.1386641392758884e+02 Gflops
        - Effective Bandwidth:      1.0265587868524264e+04 Gbps
        - Relative Error (l2):      0.0000000000000000e+00
        - PASSED

==================
Finished execution
==================
```

As you can see above, the Saxpy execution time was significantly faster than the fortran execution time. The speedup using the saxpy version is around 26x. Quite the improvement! This is running on the ampere cards.

- Theoretical maximums comparison

As you can see in the previous section, the Computational reate is around 213 Gflops, and the effective bandwidth is 1206 Gbps. This is for the saxpy implementation running on the ampere card. Using the information from Week 2, assignment 2, we can see the GPU specifications:

```
CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA A100-SXM4-40GB"
  CUDA Driver Version / Runtime Version          11.7 / 11.7
  CUDA Capability Major/Minor version number:    8.0
  Total amount of global memory:                 40390 MBytes (42352050176 bytes)
  (108) Multiprocessors, (064) CUDA Cores/MP:    6912 CUDA Cores
  GPU Max Clock rate:                            1410 MHz (1.41 GHz)
  Memory Clock rate:                             1215 Mhz
  Memory Bus Width:                              5120-bit
  L2 Cache Size:                                 41943040 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        167936 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 5 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 129 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.7, CUDA Runtime Version = 11.7, NumDevs = 1
Result = PASS
```

We are performing calculations on a NVIDIA A100-SXM4-40GB. Using this data, and https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/ we can begin comparing our output to the capabilities of the device. Beginning with the effective

bandwidth of 1206 Gbps, this is nowhere near the limit of 1555GBps. The computational rate at 213 Gflops is also quite below the theoretical maximum. The size of the vectors is set to 10,000,000 in the code. This is likely where the performance limitation is coming from, since n is used in all the calculations. Though, if I increase the size to 100,000,000 the performance is very similar. Something else is causing the performance limitation… I want to say that the limitation here is memory management. We are sending a whole lot of data through to not do a lot of calculations (y = y + a * x). The way memory being sent to the device is likely inefficient, which is leading to less performance on the SM's. In my code, I call saxpy 1000 times in my for loop. I am wondering if this is where the performance limitation lies. Since we are pinging the saxpy call so many times in a row, it may have a ton of data to deal with and just can't keep up.

Appendix 1, test.cpp

```cpp
#include <saxpy.h>
#include <Timer.hpp>

#include <cuda_runtime.h>
#include <helper_cuda.h>

#include <cmath>
#include <cstdio>
#include <vector>

// let the compiler know we want to use C style naming and
// calling conventions for the Fortran function
//
// for more details
//    - https://docs.oracle.com/cd/E19422-01/819-3685/11_cfort.html
extern "C" void saxpy_ (
  int   *   n,
  float *   alpha,
  float *   dev_x,
  int   *   incx,
  float *   dev_y,
  int   *   incy
);

double relative_error_l2 (
  int n,
  float  * y_reference,
  int inc_y_reference,
  float * y_computed,
  int inc_y_computed
);

int main (int argc, char ** argv) {


  // set values for the offset for x and y
  int incx = 1;
```

```
int incy = 1;

// set a size for our vectors
int n = 10000000;

// allocate vectors x and y_reference
std::vector<float> x (
  n * incx,
  0.0f
);
std::vector<float> y_reference (
  n * incy,
  0.0f
);

// initialize the vectors x and y to some arbitrary values,
// ideally student submissions should be random
for (int idx = 0; idx < n; ++idx) {
  x[idx * incx] = idx;
  y_reference[idx * incy] = n - idx;
}

// allocate device memory
float * dev_x = nullptr;
float * dev_y_computed = nullptr;
size_t byteSize_x = x.size() * sizeof(float);
size_t byteSize_y_reference = y_reference.size() * sizeof(float);
checkCudaErrors (
  cudaMalloc (
    &dev_x,
    byteSize_x
  )
);
checkCudaErrors (
  cudaMalloc (
    &dev_y_computed,
    byteSize_y_reference
  )
);

// copy input to dev_x, dev_y_computed
checkCudaErrors (
  cudaMemcpy (
    dev_x,
    x.data(),
    byteSize_x,
    cudaMemcpyHostToDevice
  )
);
checkCudaErrors (
  cudaMemcpy (
    dev_y_computed,
    y_reference.data(),
    byteSize_y_reference,
    cudaMemcpyHostToDevice
```

```
  )
);

// set values for the scalar
// ideally should be random for our test
float alpha = 1.0f; // the suffix f denotes float as opposed to double

// call the Fortran version of the saxpy
//  - note that we have to pass addresses of the nonpointer arguments
//  - note that we pre-declared the name the Fortran compiler produced above
//    (add a trailing underscore to the function name)

Timer fortranTimer;
fortranTimer.start();
for(int i = 0; i < 1000; i++)
{
  saxpy_ (
    &n,
    &alpha,
    x.data(),
    &incx,
    y_reference.data(),
    &incy
  );
}
fortranTimer.stop();
// should get an average runtime over many runs instead of the single one here
double fortranelapsedTime_ms = fortranTimer.elapsedTime_ms()/1000;

// start the timer
Timer saxpyTimer;
saxpyTimer.start();
// execute our saxpy
for(int i = 0; i < 1000; i++)
{
  saxpy (
    n,
    alpha,
    dev_x,
    incx,
    dev_y_computed,
    incy
  );
}
saxpyTimer.stop();

// get elapsed time, estimated flops per second, and effective bandwidth
double saxpyelapsedTime_ms = saxpyTimer.elapsedTime_ms()/1000;
double numberOfFlops = 2 * n;
double flopRate = numberOfFlops / (saxpyelapsedTime_ms / 1.0e3);
double numberOfReads = 2 * n;
double numberOfWrites = n;
double effectiveBandwidth_bitspersec {
  (numberOfReads + numberOfWrites) * sizeof(float) * 8 /
  (saxpyelapsedTime_ms / 1.0e3)
```

```
  };

  printf("\t- Fortran Execution Time:   %f Seconds\n", fortranelapsedTime_ms/1.0e3);
  printf("\t- Saxpy Execution Time:     %f Seconds\n", saxpyelapsedTime_ms/1.0e3);
  printf("\t- Computational Rate:       %20.16e Gflops\n", flopRate / 1e9);
  printf("\t- Effective Bandwidth:      %20.16e Gbps\n", effectiveBandwidth_bitspersec / 1e9);

  // copy result down from device
  std::vector<float> y_computed (
    y_reference.size(),
    0.0f
  );
  checkCudaErrors (
    cudaMemcpy (
      y_computed.data(),
      dev_y_computed,
      byteSize_y_reference,
      cudaMemcpyDeviceToHost
    )
  );

  double relerr = relative_error_l2 (
    n,
    y_reference.data(),
    incy,
    y_computed.data(),
    incy
  );

  // output relative error
  printf("\t- Relative Error (l2):      %20.16e\n", relerr);

  if (relerr < 1.0e-7) {

    printf("\t- PASSED\n");

  }
  else {

    printf("\t-FAILED\n");

  }

  return 0;

}

double relative_error_l2 (
  int n,
  float  * y_reference,
  int inc_y_reference,
  float * y_computed,
  int inc_y_computed
) {
```

```
  double difference_norm_squared = 0.0;
  double reference_norm_squared = 0.0;
  for (int idx = 0; idx < n; ++idx) {
    auto & reference_value = y_reference[idx * inc_y_reference];
    double difference {
      y_reference[idx * inc_y_reference] -
      y_computed[idx * inc_y_computed]
    };
    difference_norm_squared += difference * difference;
    reference_norm_squared += reference_value * reference_value;
  }

  return sqrt (
    difference_norm_squared / reference_norm_squared
  );

}
```