LSU EE 7700-2 **Homework 2** Solution **Due: 6 February 2013**

Problem 1: Consider the vertex transformation program presented in class:

```
#define N 5
typedef Elt_Type double;
struct Vertex { Elt_Type a[4]; };
struct App {
  int num_threads;
  Elt_Type matrix[4][4];
  int array_size;
  Vertex *v_in, *v_out;
} app;

void thread_start(void *arg) {
  const int tid = (ptrdiff_t) arg;
  const int elt_per_thread = app.array_size / app.num_threads;
  const int start = elt_per_thread * tid;
  const int stop = start + elt_per_thread;

  for ( int h=start; h<stop; h++ )
    {
      Vertex p = app.v_in[h];   Vertex q;
      for ( int i=0; i<4; i++ )
        {
          q.a[i] = 0;
          for ( int j=0; j<4; j++ ) q.a[i] += app.matrix[i][j] * p.a[j];
        }
      app.v_out[h] = q;
    }
}
```

(*a*) The code above is written for 4-element vertices. Generalize the code for $n-$element vertices. Use the macro `N` defined on the first line (which is set to 5, but could be set to other values). *Hint: The solution is fairly simple, and does not involve adding any lines of code, just modifying what's already there.*

The solution is a simple change of 4's to N's:

```
// SOLUTION
#define N 5
typedef Elt_Type double;
struct Vertex { Elt_Type a[N]; };
struct App {
  int num_threads;
  Elt_Type matrix[N][N];
  int array_size;
  Vertex *v_in, *v_out;
} app;

void thread_start(void *arg) {
```

```
  const int tid = (ptrdiff_t) arg;
  const int elt_per_thread = app.array_size / app.num_threads;
  const int start = elt_per_thread * tid;
  const int stop = start + elt_per_thread;

  for ( int h=start; h<stop; h++ )
    {
      Vertex p = app.v_in[h];   Vertex q;
      for ( int i=0; i<N; i++ )
        {
          q.a[i] = 0;
          for ( int j=0; j<N; j++ ) q.a[i] += app.matrix[i][j] * p.a[j];
        }
      app.v_out[h] = q;
    }
}
```

(*b*) Compute the number of floating-point operations per vertex when $n = 5$. A multiply-add should be counted as one floating-point operation (the multiply and the add).

There will be $5^2 - 5 = 20$ multiply-adds and 5 multiplies per vertex.

(*c*) Compute the amount of data transferred per vertex when $n = 5$. Consider both single- and double-precision numbers. (The code above is written for double-precision FP numbers, which are 8 bytes. Changing the `typedef` type to `float` will convert the code to use single-precision FP numbers, which are 4 bytes.)

Each vertex is 5 elements, and so its size is 20 bytes for single precision and 40 for double precision. A vertex is both read and written, so the transfer size is 40 B single and 80 B double. This assumes that the matrix will be read from memory just once, and so its size, $20 \times 8 = 160\,\mathrm{B}$, is insignificant when the array size is thousands of vertices.

(*d*) Repeat the problem for $n-$element vertices and an $n \times n$ matrix. Assume that $n \ll$ the number of vertices.

There will be $n^2 - n$ multiply-add instructions and n multiply instructions per vertex. The data transfer size per vertex will be $2 \times 4 \times n\,\mathrm{B}$ single precision and $2 \times 8 \times n\,\mathrm{B}$ double precision, we still don't count the matrix since n is small. If n were above a certain size then some or all of the matrix would have to be read for each vertex. In that case the data transfer amount would be $(2 + n^2)4n\,\mathrm{B}$ for single precision and $(2 + n^2)8n\,\mathrm{B}$ for double precision, which is a big difference.

The "certain size" at which the matrix would have to be re-read depends on the characteristics of the GPU. For the code given above, which does not need much other storage, the matrix could be placed in shared memory which has a capacity of $48\,\mathrm{kiB}$. It could also be placed in constant memory, but that size is only $8\,\mathrm{kiB}$.

Problem 2: The NVIDIA GTX 690, a GPU meant for home use, has the following specifications:

- Memory bandwidth: 384 GB/s.
- Single-precision computation rate of 2.8 TFLOPS (2.8×10^{12} FP operations per second).
- Double-precision rate of 117 GFLOPS (yes, less than a 10th the single-precision rate).

(For the sake of simplicity in this problem a multiply/add instruction (FMADD) is counted as one floating-point operation.)

Consider the code from the previous problem.

(a) Based on these numbers determine the maximum number of 5−element single-precision vertices per second that the GTX 690 can process.

First lets consider the FP operation throughput. The routine needs 25 FP operations per vertex. Based on the 690's FP rate of 2.8×10^{12} SP FP operations per second, we cannot compute faster than $2.8 \times 10^{12}/25 = 112 \times 10^9$ vertices per second.

Next consider memory bandwidth. That limits us to $\frac{384\,\text{GB/s}}{40\,\text{B}} = 9.6 \times 10^9$ vertices per second.

So, considering both we are limited to 9.6×10^9 vertices per second (of course we need to use the lower number).

(b) Repeat the problem above for double-precision elements.

Based on the double precision operation rate we are limited to $117 \times 10^9/25 = 4.68 \times 10^9$ vertices per second. Based on memory bandwidth we are limited to $\frac{384\,\text{GB/s}}{80\,\text{B}} = 4.8 \times 10^9$ vertices per second. Considering both, we are limited 4.68×10^9 vertices per second, this time we are limited by FP capability.

(c) At what size vertex (value of n) will both the floating point computation rate and data transfer rate both equally limit the computation rate for the single-precision vertex program?

The limit due to the FP operation rate is $2.8 \times 10^{12}/n^2$ vertices per second. The limit due to memory bandwidth is $384\,\text{GB/s}/(2 \times 8 \times n)$ vertices per second. Solving $2.8 \times 10^{12}/n^2 = 384\,\text{GB/s}/(2 \times 4 \times n)$ for n yields a limit of $n = 58\frac{1}{3}$.

What this means is that if $n < 58\frac{1}{3}$ you'd need a GPU with more memory bandwidth to improve performance (assuming that your code did bump up against the limit). If $n > 58\frac{1}{3}$ you'd need a GPU with more FP performance.

Don't forget that these are limits and even coming close to them requires skill, if it is possible at all.

(d) Consider the results above. If you need the calculations performed in double precision, why shouldn't you complain to NVIDIA about the fact that the double-precision performance is less than $\frac{1}{10}$ that of single precision for smaller values of n? *Grading note: The original question was for $n = 5$, for which there was no reason to complain. See solution.*

Grading Note: When I prepared this problem I made a mistake on the double-precision computation rate, getting a value higher than the bandwidth-limited computation rate. With that mistake, the answer to this question would be "because the double-precision code is still bandwidth limited."

Since the double-precision code is *FP-limited, there would be a good reason to complain, especially given the price of CC 3.5 GPUs (such as the K20) which cost about ten times as much as CC 3.0 devices as of this writing. One might suspect that it does not really cost ten times more to make them, they are setting the price at what the market will bear.*

For $n = 4$ the computation rate limit is 7.3125×10^9 vertices per second and the memory bandwidth limit is 6×10^9 vertices per second, and so improving double-precision performance will not help the program run faster.