Dynamic Programming with CUDA — Part II Assessment

Robert Hochberg

November 10, 2012

Assessment of Student Knowledge

1. **Competency with editing, compiling and running programs** Students who complete **The Euler-Mascheroni Constant** and **Compiling with Doubles** can be assumed to have mastered the basic development cycle. But no deep knowledge of CUDA programming could be inferred.
2. **Understanding of Lookback Doubling:** A student who can answer the **Binary Addition** problem has mastered the technique of look-back doubling.
3. **Ability to Program in CUDA:** Students who can compile and run the sample code, and answer the **Integer Exploration** question has likely understood the CUDA programming model, and has shown the ability to write a kernel and perform memory transfers from the card.
4. **CUDA Mastery:** Students who can solve the **Binary Addition** and the **CUDA Global Memory** problems have likely gained enough skill to begin writing programs from scratch in the CUDA environment in a way that takes advantage of its capabilities.

Some Solutions

1. **Exercise.** Suppose the threads in a kernel use 8 bytes of shared memory each, and the system requires 12 bytes per block of threads. If you are going to be running on a GeForce GTX 260, then what would be a good number of threads per block for this kernel? (See Appendices A and F of the Cuda C Programming Guide [2].) Repeat the calculation if instead of 8 bytes per thread we need 48 bytes per thread.

   The GeForce GTX 260 is of compute capability 1.3 (Appendix A) and so has 16KB of shared memory per block (Appendix F). In the first, 8 bytes per thread, case, there is enough shared memory for $(16384 - 12)/8 = 2046$ *threads per block. But the maximum number of threads per block for a device of compute capability 1.3 is 512 (Appendix F). So that would be a good number to use. If each thread needs 48 bytes of memory, then we have enough space for* $(16384 - 12)/48 = 341$ *threads per block. So that might be a good number to use, but it would be worth checking experimentally whether using the next-smaller multiple of 32, that is, 320 threads per block might be better. (See the end of section 5.2 of the CUDA C Programming Guide.)*

2. **Exercise.** The program below is included with this module: `exercise.cu`. (Some non-essential lines have been deleted to make it fit on the page. The included file has everything needed to compile.) It can be compiled with `nvcc exercise.cu` and run with `./a.out`. Before running the program, decide what the program does, and then check your answer by running it.

   The program copies a 4×4 *matrix up to the CUDA device, squares it, and then copies it back to the host. Each thread on the device is responsible for 1. Loading one entry from global memory into shared memory, 2. computing one entry in the square of the matrix, and 3. saving the result back to global memory.*

```
typedef struct{
  int a[4][4];
} Matrix;

__global__ void compute(Matrix* mIn, Matrix* mOut){
  __shared__ Matrix m1;
  int row = threadIdx.x / 4;
  int col = threadIdx.x % 4;
  m1.a[row][col] = (*mIn).a[row][col];

  int i;                          /* counter */
  int sum = 0;
  for(i = 0; i < 4; i++)
    sum += m1.a[row][i] * m1.a[i][col];
  (*mOut).a[row][col] = sum;
}

int main(void){
  int i, j;                       /* counters */
  Matrix *m1, *m2;
  m1 = (Matrix*)malloc(sizeof(Matrix));
  for(i = 0; i < 4; i++)
    for(j = 0; j < 4; j++)
      (*m1).a[i][j] = random() % 10;

  Matrix *mDevIn, *mDevOut;
  cudaError_t err = cudaMalloc(&mDevIn, sizeof(Matrix));
  err = cudaMalloc(&mDevOut, sizeof(Matrix));

  err = cudaMemcpy(mDevIn, m1, sizeof(Matrix), cudaMemcpyHostToDevice);
  compute <<< 1, 16 >>> (mDevIn, mDevOut);

  m2 = (Matrix*)malloc(sizeof(Matrix));
  err = cudaMemcpy(m2, mDevOut, sizeof(Matrix), cudaMemcpyDeviceToHost);
}
```

3. **The Euler-Mascheroni Constant.** Let's do a quick warm-up. The sum $1+1/2+1/3+1/4+1/5+\cdots+1/n$ is approximately equal to $\ln n$, the natural log of n. As n gets large, the difference between the sum and $\ln n$ approaches the *Euler-Mascheroni* constant γ, which is about 0.5772156649. We will use our recurrence to evaluate the sum, which we will then compare to the natural log.

   Consider the file `recursion.c`. Lines 305-311 in `main()` set up the recurrence:

```
// Set up the initial values {a[-2], a[-1], a[0], 1}
float myInit[] = {1.0, 0.0, 0.0, 1.0};
cudaMemcpyToSymbol(init, myInit, 4*sizeof(float));

// Set up the initial recursion {c1, c2, c3, d}
float myRec[] = {1.0, 0.0, 0.0, 1.0};
cudaMemcpyToSymbol(rec, myRec, 4*sizeof(float));
```

   The initialization given above corresponds to the recurrence $a_i = a_{i-1} + 1$. This is saved in `__constant__` memory, as indicated by `cudaMemcpyToSymbol`. Since about 8192 bytes of constant memory can be cached, all threads will have fast access to this recurrence. (See Section 5.3.2.4 of the Cuda C Programming Guide [2] for more on constant memory.) The program as provided with this module has "`rule[threadIdx.x].a[3][0] = (float)1.0/idx;`" on line 147. This gives each thread its own value for "d" so that the recursion rule is $a_i = a_{i-1} + 1/i$. (Note that if the recursion depends on i, then we must allow each thread to build its own rule, using its own index. This takes place in the `initializeRules()` kernel function.) Compile the program `recursion.cu` as follows:

```
nvcc -o recur -DNUMELTS=1000 -DthreadsPerBlock=255 recursion.cu
```

   (The `-D` options `#define` the number of elements to inspect (`NUMELTS`) to be a thousand, and to use 255 threads per block.) Then type `./recur` to run the program.

   You should see a printout of the first 1000 partial sums: 0, 1, 1.5, 1.83333, 2.083333, ... The last value is $1 + 1/2 + 1/3 + \cdots + 1/999$ which should be about $\ln 999$. Compare this with the actual value of $\ln 999$ to get our first estimate of the Euler-Mascheroni constant. Experiment by re-compiling with more elements, and seeing how far you can push this approximation. Note that

you may want to modify the `printout()` function to reduce output.

The included file "em.cu" contains a solution to this problem.

4. **Compiling with Doubles.** In a real scientific application where more precision was needed than 32-bit floats could provide, doubles (64 bits) or perhaps arbitrary precision numbers would be used. Unfortunately, devices of compute capability 1.2 or lower do not support native doubles. Included with this module is a file `recursionDouble.cu` which, if compiled with the `-arch=sm_13` flag (see Section 3.1.3 of the CUDA C Programming Guide) will use doubles instead of floats (but only on devices of compute capability at least 1.3) to compute the Euler-Mascheroni constant.

   (a) How many threads per block are you able to have now?

   (b) If you run the code, do the partial sums look any more accurate when doubles are used than when floats are used?

   (c) One thing often overlooked is the error that creeps in whenever we use computers to do arithmetic on real numbers, such as $1/3$ or π. Replace line 147 of `recursionDouble.cu` to read

   ```
   rule[threadIdx.x].a[3][0] = abs(1-((double)1.0/idx)*idx);
   ```

   All of these values are "mathematically" equal to 0, but if we add them all up (as we added $1/i$ in the previous problem) you can see how the error accumulates. Compile `recursionDouble.cu` without the `-arch=sm_13` flag to force it to use floats, and see how large the error is when adding 1000 elements.

   (d) How would you expect things to turn out if we used doubles instead of floats? Repeat the exercise with `recursionDouble.cu`, compiled with the `-arch=sm_13` flag, to see if your intuition is correct. (Of course, this assumes that you have a device of compute capability at least 1.3 on which to try this.)

   (e) Finally, repeat Problem 1 using `recursionDouble.cu`. Do you notice any difference in your approximations of γ?

5. **Integer Exploration.** If we tried to compute the Fibonacci numbers with floats, or even doubles, we would not be able to get very far. Computation with doubles fails to get the exact value before even the 100th term, and a

double can't even hold the 1500th term because it exceeds the maximum size of a double.

The program `recursionInt.cu` uses `int`s instead of floats or doubles, and makes use of a `MOD` macro which can be defined at compile time. The program will then compute all values modulo `MOD`, which no value will ever exceed. Modular arithmetic is such that if we are computing only sums and products, then we may reduce modulo `MOD` as often as we wish during our computation, and our end result will be the same. This conveniently allows us not to have to otherwise modify our algorithm. If we compile the program to run mod 7:

```
nvcc -DMOD=7 -DNUMELTS=100 -o recurInt recursionInt.cu
```

then we can see the first 100 terms of the Fibonacci sequence mod 7. The sequence begins: 0, 1, 1, 2, 3, 5, 1, 6, 0, 6, 6, 5, 4, 2, 6, 1, 0, 1, 1, 2, 3, ... Notice that it repeats after the first 16 terms, so that the sequence will be periodic with period 16.

(a) After the `while` loop in `main` has ended, all of the threads have computed their values, and they are sitting in global memory, in an array of `Rule` structures pointed to by `devRules`. Write a new kernel (`__global__`) function that can be called at this point that will inspect these values and find the period. This can always be determined by finding the first re-ocurrence of the consecutive values "0, 1". *This is included with the module as fib.cu. It will also print out all 100000 elements, but you can change the "printout" function in fib.cu to do this differently. The reason for the "arch" flag is that the code uses an* `atomicMin` *function which is not available on devices of compute capability less than 1.1, so we tell the compiler that we'll be on compute capability at least 1.1. If you're on higher compute capability, you can use* `arch=sm_12`*, etc... to let the compiler know. Note that this code will not run on devices of compute capability 1.0.*

The way this code works is that we create a thread for each element, and each thread checks to see if its own element has value "0" while the element to its right has value "1." Each thread block has 512 threads, but only the first 511 threads in each block perform this check. The 512th thread is simply in charge of reading a value from the array in global memory into shared memory, so that the 511th thread has access to it.

(b) As part of the preceding problem, you need to devise a way to get a value

(the period) off of the device. How did you solve this problem? *In our code, we created a variable* `int hostPeriod` *which resided on the host, and a variable* `int devPeriod` *which resided on the device via* `cudaMalloc`. *We used* `cudaMemcpy` *to read the period off of the device after it had been computed.*

(c) Another problem is that you may have many blocks all trying to shove an answer into the same location in memory, if they have discovered an occurrence of "0, 1" in the `devRules` array. But only the first occurrence should be returned. How did you solve this problem? *This code uses* `atomicMin` *to solve this problem. For this reason, however, the code does not run on devices of compute capability 1.0.*

(d) The period of the Fibonacci numbers mod 49 is 112. It seems that for all primes p, the period of the Fibonacci numbers mod p^2 is strictly greater than the period mod p. Nobody knows if this is the case for all primes, though. Check it out for some primes on your own, and see if you can find a counter-example. If you do, make sure you show a number theorist at your school, because everyone's wondering if there is such a counter-example.

Here are the periods for all primes less than 100 and their squares. The file `fibRep.cu` *and the script* `manyFib` *can be used to explore this question, by typing* `./manyFib`

Prime p	*period mod p*	*period mod p^2*
2	*3*	*6*
3	*8*	*24*
5	*20*	*100*
7	*16*	*112*
11	*10*	*110*
13	*28*	*364*
17	*36*	*612*
19	*18*	*342*
23	*48*	*1104*
29	*14*	*406*
31	*30*	*930*
37	*76*	*2812*
41	*40*	*1640*
43	*88*	*3784*
47	*96*	*1504*
53	*108*	*5724*
59	*58*	*3422*
61	*60*	*3660*
67	*136*	*9112*
71	*70*	*4970*
73	*148*	*10804*
79	*78*	*6162*
83	*168*	*13944*
89	*44*	*3916*
97	*196*	*19012*

(e) Try some experiments with a modified Fibonacci recurrence, such as $a_i = a_{i-1} + a_{i-2} + 1$. What is the period of this sequence mod p for the primes under 50?

Here is a table of the periods mod x, for all x from 2 to 99, and their squares. Notice that the periods for prime values and their squares don't change! The code for this problem can be found in `fibRepPlus1.cu`*, and can be run explored with the shell script* `manyFibPlus1`*. Note that the shell script may need to be made executable by the user:*

```
chmod u+x ./manyFibPlus1
```

x	$period \bmod x$	$period \bmod x^2$	x	$period \bmod x$	$period \bmod x^2$	x	$period \bmod x$	$period \bmod x^2$
2	3	6	35	80	2800	68	36	1224
3	8	24	36	24	216	69	48	1104
4	6	24	37	76	2812	70	240	8400
5	20	100	38	18	342	71	70	4970
6	24	24	39	56	2184	72	24	864
7	16	112	40	60	2400	73	148	10804
8	12	384	41	40	1640	74	228	8436
9	24	216	42	48	336	75	200	15000
10	60	300	43	88	3784	76	18	1368
11	10	110	44	30	1320	77	80	6160
12	24	24	45	120	5400	78	168	2184
13	28	364	46	48	1104	79	78	6162
14	48	336	47	64	1504	80	120	9600
15	40	600	48	24	384	81	216	17496
16	24	768	49	112	5488	82	120	4920
17	36	612	50	300	7500	83	168	13944
18	24	216	51	72	1224	84	48	336
19	18	342	52	84	2184	85	180	15300
20	60	600	53	108	5724	86	264	11352
21	16	336	54	72	1944	87	56	4872
22	30	330	55	20	1100	88	60	5280
23	48	1104	56	48	672	89	44	3916
24	24	192	57	72	1368	90	120	5400
25	100	2500	58	42	1218	91	112	1456
26	84	1092	59	58	3422	92	48	1104
27	32	1944	60	120	600	93	120	3720
28	48	336	61	60	3660	94	96	4512
29	14	406	62	30	930	95	180	17100
30	120	600	63	48	3024	96	48	1536
31	30	930	64	288	6144	97	196	19012
32	48	1536	65	140	9100	98	336	16464
33	40	1320	66	120	1320	99	120	11880
34	36	612	67	136	9112	100	300	15000

(f) Finally, answer the same question for the recurrence $a_i = a_{i-1} + a_{i-2}$, where $a_0 = 1$ and $a_1 = 1$. Is anything periodic here?

This question is much more interesting, because you can't simply search for a repeat of "1, 1" to find the period. For example, the sequence generated mod 5 begins: 1, 1, 4, 3, 1, 4, 1, 2, 1, 2, 3, 1, 1, 0, 0, 0, 1, 3, 2, ... Notice that the first re-occurrence of "1, 1" is followed by a "0", not a "4". Eventually there is another "1, 1, 4" in this sequence, and that is where you can determine the period. In the provided code, `fibRepPlusi.cu`*, the kernel function* `findPeriod()` *contains this test for discovery of the period:*

```
if(threadIdx.x < blockDim.x - 1 && idx > 1
   && fibValues[threadIdx.x] == init[2]
   && fibValues[threadIdx.x + 1] == init[2] + init[1] + 1
   && (idx % MOD) == 0)
```

By checking that `idx% MOD == 0` *we are checking that the index (i) has returned to 0 modulo the modulus, in addition to the terms returning to "1, 1". What I found truly surprising is that eventually, under all moduli tested, the subsequence "1, 1, 4" did eventually re-occur. Perhaps there is a theorem here? I haven't tried to prove it yet. (But you can prove that the re-occurrence of "1, 1, 4" is equivalent to the sequence being purely periodic, right?)*

x	*period* mod x	*period* mod x^2	x	*period* mod x	*period* mod x^2	x	*period* mod x	*period* mod x^2
2	6	12	35	560	19600	68	612	41616
3	24	72	36	72	1296	69	1104	76176
4	12	48	37	2812	104044	70	1680	58800
5	20	100	38	342	12996	71	4970	352870
6	24	72	39	2184	85176	72	72	5184
7	112	784	40	120	4800	73	10804	788692
8	24	192	41	1640	67240	74	8436	312132
9	72	648	42	336	7056	75	600	45000
10	60	300	43	3784	162712	76	684	51984
11	55	1210	44	660	29040	77	6160	474320
12	24	144	45	360	16200	78	2184	85176
13	364	4732	46	1104	25392	79	6162	486798
14	336	2352	47	1504	70688	80	240	19200
15	120	1800	48	48	2304	81	648	52488
16	48	768	49	784	38416	82	4920	201720
17	612	10404	50	300	7500	83	13944	1157352
18	72	648	51	1224	20808	84	336	7056
19	342	6498	52	1092	113568	85	3060	260100
20	60	1200	53	5724	303372	86	11352	488136
21	336	7056	54	216	5832	87	4872	423864
22	330	7260	55	220	12100	88	1320	116160
23	1104	25392	56	336	9408	89	3916	348524
24	24	576	57	1368	25992	90	360	16200
25	100	2500	58	1218	70644	91	1456	132496
26	1092	14196	59	3422	201898	92	1104	25392
27	216	5832	60	120	3600	93	3720	345960
28	336	2352	61	3660	223260	94	4512	212064
29	406	11774	62	930	57660	95	3420	1624500
30	120	1800	63	1008	63504	96	96	9216
31	930	28830	64	192	12288	97	19012	1844164
32	96	3072	65	5460	118300	98	2352	115248
33	1320	43560	66	1320	43560	99	3960	392040
34	612	10404	67	9112	610504	100	300	30000

6. **Binary Addition.** Implement in CUDA the algorithm for binary addition of numbers with millions of bits. Even on a device with only 256 megabytes of global memory you can store two addends and a sum, each with 670 million

bits, with room left over. Have each thread (processor) be responsible for one `unsigned int` (32 or 64 bits) worth of data. This will allow you to compute in a single instruction the sum of all the bits in the `unsigned int` at once, taking care of all the carries internally, and producing one carry.

7. **A Tree-Full of Heads.** The implementation presented here starts with many blocks of threads, each with a rule for updating from the thread adjacent to it, and at some point has linked all of their heads. At this point we may copy those heads to a separate part of global memory, so that each head has a rule for updating from the head adjacent to it. If these all lie in a single block of threads, then we can use `propagate`, as opposed to many calls to `doubleHeadLookback`, to give a value to all heads. If those heads comprised several blocks, then we can recursively iterate the process, taking the heads of *those* blocks, linking them, copying them to another part of global memory, etc..., iterating until all the heads lie in a single block, at which point we propagate the values in that one block. The original steps, and this modification, are shown below:

Original Algorithm

(a) Processor (thread) 0 gets a value. All others get rules looking back one thread.

(b) Threads within a block all build rules that look back to the head of the block. Those in Block 0 get values.

(c) Every head builds a rule looking back at the previous head.

(d) Head threads double their lookback until they all get values from the head of Block 0.

(e) Threads within a block all get their values from the head of their block.

Modified Algorithm

(a) Processor (thread) 0 gets a value. All others get rules looking back one thread.

(b) If the threads all fit into one block, propagate the value at the head to the rest of the threads

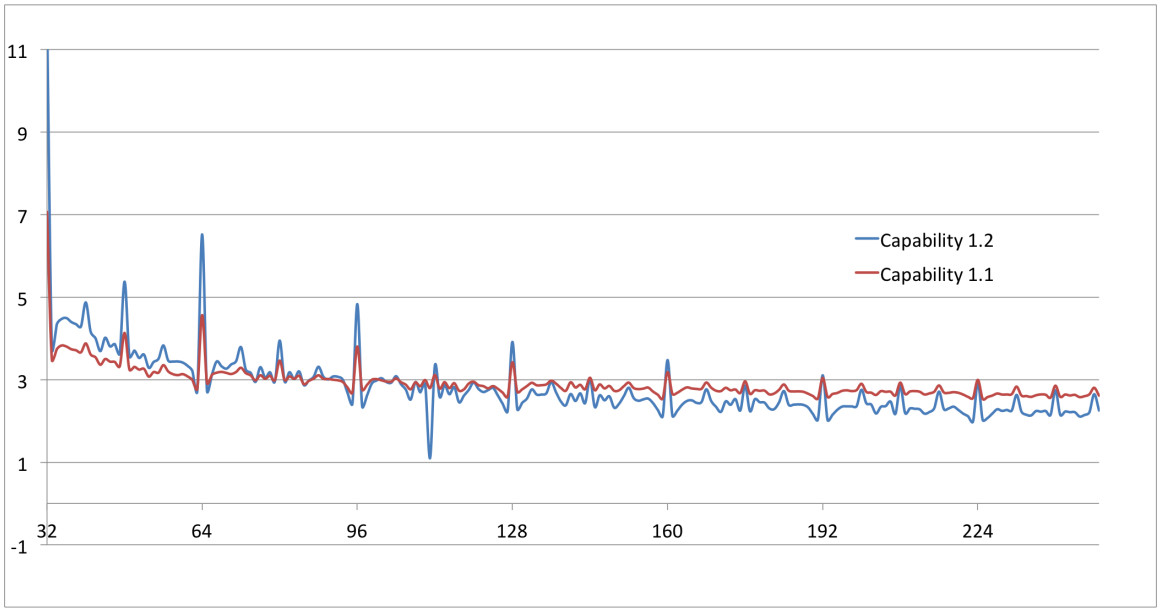
(c) If they don't fit into one block, have each head build a rule looking back to the previous head, fill new blocks with just the heads, and solve this problem by recursively going back to step 1.

(d) When we exit a recursive call, have the head of each block propagate its value to the block that it was originally from. Threads per block should be a multiple of warp size to avoid wasting computation on under-populated warps and to facilitate coalescing.

Code this modified algorithm, then run tests to see if there is any difference in performance compared to the original algorithm.

8. **CUDA Global Memory.** On page 48 of the CUDA C Best Practices Guide [1] we read, "*Threads per block should be a multiple of warp size [warp size = 32] to avoid wasting computation on under-populated warps and to facilitate coalescing.*" So we do a bit of experimentation. Figure **??** shows the running times for `recur` computing on a million elements. On the horizontal axis we vary the number of threads per block, from 32 to 255, and the vertical axis shows running time in seconds. It seems that the *worst* running times occur when the number of threads per block is a multiple of 32. Some investigation reveals that the culprit is the kernel function `copyHeadDataFromTemp`. Look over section 3.2.1 of the CUDA C Best Practices Guide to see if *non-coalesced memory accesses* are the cause. And look over Section 3.2.2 of the CUDA C Best Practices Guide to decide if *memory bank conflict* could be the culprit. It may be both, one or neither.

   It's neither. All of the `Rule` *data structures are 64 bytes, and they lie in memory allocated by* `cudaMalloc` *so that they are all 64 byte-aligned. They are optimally positioned for coalesced reads and writes. Also, the* `copyHeadDataFromTemp` *kernel function makes no use of shared memory, so there are no memory bank conflicts. The problem has to do with the way global memory is physically structured. As does shared memory, global memory consists of some number of "banks." And when too many writes are being made to the same bank, the data ends up waiting for its turn to be written to the memory. This phenomenon is called "partition camping." A paper entitled* Bounding the Effect of Partition Camping in GPU Kernels *is included with this module.*


11
9
7
5
3
1
-1
32
64
96
128
160
192
224
Capability 1.2
Capability 1.1

Bibliography

[1] The NVIDIA Corporation. *The CUDA C Best Practices Guide v4.0*. NVIDIA Corporation, 2011.

[2] The NVIDIA Corporation. *The CUDA C Programming Guide v4.0*. NVIDIA Corporation, 2011.