Dynamic Programming with CUDA — Part II

Robert Hochberg

November 10, 2012

Contents

Chapter 1

Introduction

1.1 Overview

This module is largely stand-alone. It is "Part II" only in the sense that it does not contain the overview of dynamic programming seen in Part I, and does not recapitulate the introduction to CUDA. We will continue to refer the reader to various NVIDIA references where appropriate, particularly the NVIDIA CUDA C Programming Guide [5], and the CUDA API Reference Manual [3], and where we introduce new CUDA-specific ideas, will linger a bit longer by way of introduction. The algorithms described here are completely independent of Part I, so that a reader who already has some familiarity with CUDA and dynamic programming may begin with this module with little difficulty.

1.1.1 Organization of the Module

The emphasis of this module is the notion of "lookback doubling." This technique enables cooperating processors to combine their efforts to achieve exponential speedups in some contexts, including some dynamic programming contexts. In Chapter 2, we will focus on two problems:

- **Parallel Binary Addition:** Straightforward addition of two n-bit binary numbers requires n steps, even on n processors, because we need to propagate the carries. We will show how to use "lookback doubling" to compute the sum in $\log_2 n$ steps on n processors.

- **Linear Recurrences:** We will consider sequences generated by recurrences of the form $a_n = c_1 a_{n-1} + c_2 a_{n-2} + c_3 a_{n-3} + d$. Such sequences include the Fibonacci and Lucas sequences. Inspecting the first n terms of this sequence on a single processor would require time proportional to n. We show how to use "lookback doubling" on n processors to inspect the first n elements in time proportional to $\log n$.

We will then consider the issues involved in solving these problems on a General Purpose Graphics Processing Unit (GPGPU)-enabled computer, such as the graphics cards found in many of our computers today. GPGPU cards have dozens or even hundreds of cores, which can run threads in a highly parallel fashion. In one way of thinking about parallel algorithms, we may code as if we have millions of processors available, create a thread to run on each processor, and then have a GPGPU run these threads in batches of dozens or hundreds, according to how many cores the GPGPU has. The technical aspects of setting up these threads, synchronizing them as needed, loading them onto the GPGPU, managing memory, running the threads and reading the results, is the subject of Chapter 4.

Chapter 2

Developing the Parallel Algorithm

2.1 Parallel Addition

We begin with a classic problem in parallel computing, that of adding together two binary numbers. Figure 2.1 shows the addition of two binary numbers with the carry bits shown in red above the addends.

Figure 2.1: Adding two binary numbers, showing the carries in red.

How many steps does it take to add these two numbers together? When we add them by hand, it takes as many steps as there are bits in the larger number. Adding from right-to-left, we add, carry, add, carry, add, carry, etc..., with each add/carry being a single step. But what if we had a processor for each bit of the summand? For convenience, let us suppose the processors are numbered 0, 1, 2, ..., with the ith processor adding the bits in the 2^i place. Then in a single step, each processor could perform its add and we'd be done, right? Well, not quite. Processor 0 (the rightmost processor) can perform its addition, but Processor 1 (second from the right) can't

perform its addition until it knows whether Processor 0 will produce a carry or not. In general, processor i must wait for processor $i-1$ to perform its addition and produce, or not, a carry.

So perhaps there is no benefit to having many processors. It seems that it will still take n steps to compute a sum having n bits. Fortunately, this is not the case. When designing algorithms for multiple processors it is often necessary to devise genuinely new solutions to problems, to develop brand new algorithms that are very different from their non-parallel counterparts.That's what we'll do here.

Let us consider Processor 15 in the addition example of Figure 2.1. Instead of just waiting for Processor 14 to announce whether or not it will generate a carry, it can perform *two* additions, one in the case that there is a carry, called the *carry-sum*, and one in the case that there is not, called the *no-carry-sum*. Now it will be ready with its sum as soon as it finds out whether Processor 14 has a carry or not. You might not think that's much of an improvement, and you'd be right if only Processor 15 did this. But suppose that Processor 14 had also computed two sums, one for each case of Processor 13 generating a carry or not. If it conveys this information to Processor 15, then *both* Processors 14 and 15 will be able to compute their sums as soon as Processor 13 declares whether or not it will produce a carry. For Processor 15's thinking will go something like this: "If Processor 13 produces a carry, then so will Processor 14, in which case I use my *carry-sum*, and if Processor 13 does not produce a carry, then neither will Processor 14, in which case I will use my *no-carry-sum*."

Suppose now that every processor did these two steps: produced two sums depending on its neighbor's carry, and used its neighbor's information to decide which sum to use once it knows about the carry produced by its neighbor two processors away. Now each processor is able to produce its sum immediately once its neighbor *two* places away knows whether it will produce a carry or not.

This is the big idea: After 1 step, each processor could get its result by "looking back" a distance of one processor. After the next step, each processor could get its result by "looking back" a distance of two processors. We will iterate this process, so that after each step, each processor can get its result by "looking back" twice as far. This way, each processor can have its final result after $\log_2 n$ steps, as we'll see.

Let us consider Processor 15 again. At this point it can produce its sum as soon as it knows whether Processor 13 will produce a carry or not. And Processor 13 can produce its sum (and carry) as soon as it knows whether Processor 11 will produce a carry or not. By putting this information together, Processor 15 will be able to

produce its sum as soon as it knows whether Processor 11 will produce a carry or not. In this way, Processor 15 can effectively "look back" a distance of four processors. Since Processor 11 did the same thing, and can now "look back" four processors, on the next step we can combine these "look-backs" so that Processor 15 can "look back" eight processors. And so on.

Let's look at what's happening at the lower end of the processor list. Processor 0 can compute its sum and carry immediately, since it knows there will be no carry figuring into its sum. After the second step, when Processor 1 is looking back a distance of one processor, it can compute its sum and carry, since Processor 0 will have produced its carry after step 1. After the third step, Processor 2 *and* Processor 3 can produce their sums and carries, because they are looking back to Processors 0 and 1, respectively, which will have produced their carries after step 2. In this fashion, after each step, the number of processors that can compute their final sums doubles.

Let's formalize these ideas: For each bit in the sum, let us have a data structure as shown in the figure below.

```
struct sumStructure {
   bool sumIfNoCarry;    /* Sum and carry bits in the     */
   bool carryIfNoCarry;  /* two cases that the bits we    */
   bool sumIfCarry;      /* are looking back to do or     */
   bool carryIfCarry;    /* don't have carries.           */
   bool value;           /* True if we have a sum,
                              false if just looking back. */
   bool carry;           /* If we've computed a sum,
                              was it the carry case?      */
}
```

The first two fields contain the sum and carry in the case that the processor we are looking back to does not produce a carry, and the next two fields hold the sum and carry in the case that the processor we are looking back to does produce a carry. We will use "1" for "true" and "0" for "false," as usual. In the pictures that follow, we'll depict this data structure as shown in Figure 2.2. We could have added an `int lookbackDistance` field to our data structure, showing how far back each Processor was looking. But in our implementation this value will be the same for all processors, so we'll keep it as a global that all processors have access to.

We will store our `sumStructure`s in an array `sumStructure ss[N]`, where `N` is the

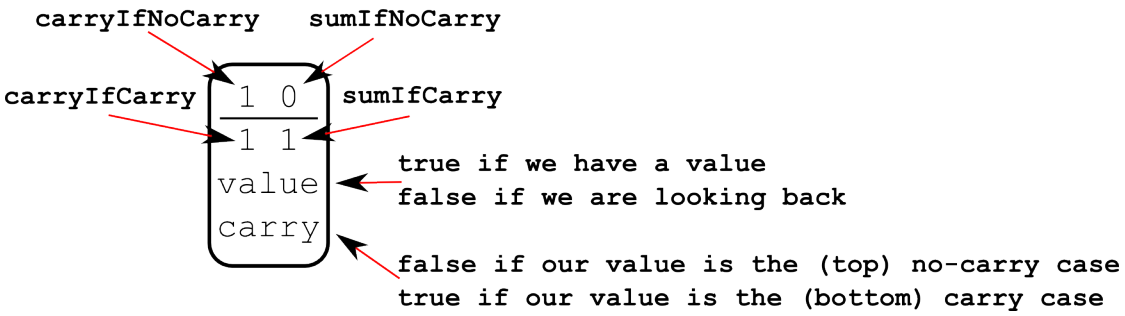

Figure 2.2: Pictorial depiction of the sumStructure data structure.

number of bits in the sum, and hence the number of processors. At each step, a processor may double the distance it looks back by calling the `oneStep()` method shown in Figure 2.3

Exercise: Justify the doubling rule given by the last two "if" clauses in the `oneStep()` method shown in Figure 2.3.

The complete algorithm for parallel binary addition can be found in Figure 2.4.

Notice that as long as we have one processor per bit in the sum, the number of steps is on the order of the log of N rather than N itself. This exponential speedup shows the power of developing new algorithms as opposed to implementing old algorithms on more processors.

In the next section we will consider a problem whose solution involves dynamic programming, and whose parallel implementation uses the ideas just considered for binary addition. We will develop a CUDA-based solution for that problem, but leave the CUDA-based implementation of binary addition as an exercise for the reader. Its solution would closely mimic (and be rather simpler than) that given for the problem in the next section.

2.2 Linear Recurrences

The Fibonacci sequences $F_0, F_1, F_2, \ldots$, which begins $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ is defined by the recurrence: $F_0 = 0, F_1 = 1, and F_n = F_{n-1} + F_{n-2}$ for each $n \geq 2$. In this section we will explore similar recurrences, namely those of the form: $a_n =$

```
/* Before we run this, each element either has a value, or knows
 * how to get its value once the processor that is lookbackDistance
 * away knows its values.
 * When this method returns, the i'th processor will be looking back
 * twice as far as before. That is, it will be able to compute its
 * own sum and carry as soon as the processor it's looking back at
 * knows if it will have a carry or not.
 */
void oneStep(sumStructure* ss, int myIndex){
   int other = myIndex - lookbackDistance;

   if(value == true) return;     // if we already know our value

   // Are we looking back to a processor that knows its value?
   if(ss[other].value == true){
      if(ss[other].carry == false)
         ss[myIndex].carry = ss[other].carryIfNoCarry;
      else
         ss[myIndex].carry = ss[other].carryIfCarry;
      ss[myIndex].value = true;      // I have a value now
      return;
   }

   // Otherwise, we will double our lookback
   // First check if the "other" produces a carry in all cases
   if(ss[other].carryIfNoCarry == true){  // Copy bottom row to top
      ss[myIndex].carryIfNoCarry = ss[myIndex].carryIfCarry;
      ss[myIndex].sumIfNoCarry   = ss[myIndex].sumIfCarry;
   }
   // Then check if the "other" produces a carry in no case
   if(ss[other].carryIfCarry == false){  // Copy top row to bottom
      ss[myIndex].carryIfCarry = ss[myIndex].carryIfNoCarry;
      ss[myIndex].sumIfCarry   = ss[myIndex].sumIfNoCarry;
   }
}
```

Figure 2.3: Composing lookback rules to double the lookback distance.

```
Let N = the number of processors
      = largest number of bits the sum might have
      = 1 + number of bits in the larger addend
Let A[N] be the array of bits for the first addend
    B[N] be the array of bits for the second addend
// (We pad A and B with as many initial 0s as needed)

// Initialize the ss[] data structures
Each processor i Do:
    ss[i].sumIfNoCarry   = A[i] XOR  B[i]
    ss[i].carryIfNoCarry = A[i] AND B[i]
    ss[i].sumIfCarry     = A[i] ==  B[i]
    ss[i].carryIfCarry   = A[i] OR  B[i]
    ss[i].value          = false
    ss[i].carry          = false
// Now all the processors are looking back distance 1
// No processor has a value

// Give a value to processor 0
ss[0].value = true
ss[0].carry = false // just to be explicit here

// Now iterate until every processor has a value
Do log_2(N) times:
    Each processor i Do:
        oneStep(ss[])
// Now every processor has a value

// Finally, put the answers in C
Each processor i Do:
    if ss[i].carry == false:
        C[i] = ss[i].sumIfNoCarry
    else
        C[i] = ss[i].sumIfCarry
```

Figure 2.4: The complete algorithm for parallel binary addition

```
/* We assume that a_0, a_1 and a_2 are given
 * and that c_1, c_2, c_3 and d are known
 */
double findTerm(k){
   if(k == 0) return a_0;
   if(k == 1) return a_1;
   if(k == 2) return a_2;
   return c_1*findTerm(k-1) + c_2*findTerm(k-2) +
          c_3*findTerm(k-3) + d;
}
```

Figure 2.5: Using recursion to find sequence values given by a recurrence.

$c_1 a_{n-1} + c_2 a_{n-2} + c_3 a_{n-3} + d$ for $n \geq 3$, where c_1, c_2, c_3 and d are real constants, and initial values are given for a_0, a_1 and a_2.

The most casual search for "Fibonacci Numbers" will reveal a breathtaking collection of applications for this sequence of numbers, including cryptography and bar codes [1], quasi-periodic formations in flames, flowers and waves [8], searching [7] and, of course, mathematics. These and many more applications can be found for the broader class of recurrences described above.

Here, we will want to consider two types of questions about these recurrences: **Evaluate:** What is the kth term? **Search:** Among the first so-many terms, $a_0, a_1, \ldots$, which value of a_i gives the best answer for our particular problem, whatever it might be.

A very simple program to compute the kth term in this sequence might look like that shown in Figure 2.5.

This very simple implementation is easy to code and returns the right answer. But the number of steps required to compute the nth term grows exponentially, and is about 1.84^n [6].

We use *memoization*, a standard technique of dynamic programming, to improve this running time. Each time we compute a value, we store the result in a lookup table of some sort. And before computing a value, we look in the table to see if we've already computed it. This "recursion with memoization" is shown in Figure 2.6.

```
/* We assume that a_0, a_1 and a_2 are given
 * and that c_1, c_2, c_3 and d are known.
 * We memoize our results in the table[] array.
 */
double findTerm(k){
   if(table[k] != -1) // -1 means "not found yet"
       return table[k];
   if(k == 0) return a_0;
   if(k == 1) return a_1;
   if(k == 2) return a_2;
   table[k] = c_1*findTerm(k-1) + c_2*findTerm(k-2) +
              c_3*findTerm(k-3) + d;
   return table[k];
}
```

Figure 2.6: Using recursion with memoization to find sequence values given by a recurrence.

2.2.1 Parallel Considerations — Rules and Values

Let us first consider the question of search: How can we search the first million (for example) terms of a sequence defined recursively by $a_n = c_1 a_{n-1} + c_2 a_{n-2} + c_3 a_{n-3} + d$? As with the case of binary addition, it might seem that it should take about a million steps, since in order to even determine term $a_{1000000}$, we need to know terms a_{999999}, a_{999998} and a_{999997}. And in order to find a_{999999} we need to find ..., and so on. Fortunately, we may develop a brand new algorithm here that allows us to solve this problem more efficiently.

We will model our algorithm here on the binary addition example discussed above, where each processor had either a **rule** or a **value**. A processor with a value was done with its computation, while a processor with a rule could either **evaluate** if it is looking back at a processor with a value, or **double** if it is looking back at another processor with a rule.

Let us suppose we have a processor for each term of the sequence. (More on dealing with the unrealistic-ness of this later.) The first step is to simplify the lookback, for if we use the recurrence as-is, then each processor needs to look back to the *three* processors preceding it, and this trifurcation complicates the step where we double

our lookback distance. It should be possible, for example, for processor P_i, looking back to P_{i-1}, to determine its value if Processor P_{i-1} has its value.

We begin to accomplish this by storing at each processor P_i a data structure consisting of the triple (a_i, a_{i-1} and a_{i-2}). Now Processor P_{i+1} may compute its triple $(x, y, z) = (a_{i+1}, a_i$ and $a_{i-1})$ from P_i via the equations:

$$\begin{aligned} x &= c_1 a_i + c_2 a_{i-1} + c_3 a_{i-2} + d \\ y &= a_i \\ z &= a_{i-1} \end{aligned} \qquad (*)$$

These equations amount to a **rule** for computing a value. A very natural way to express this rule is with a 4×4 matrix, yielding the equation below:

$$\begin{pmatrix} a_i & a_{i-1} & a_{i-2} & 1 \end{pmatrix} \begin{pmatrix} c_1 & 1 & 0 & 0 \\ c_2 & 0 & 1 & 0 \\ c_3 & 0 & 0 & 0 \\ d & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_{i+1} & a_i & a_{i-1} & 1 \end{pmatrix}$$

Notice that this requires that we augment the "value" data structure stored at each processor so that it includes a "1" in its fourth coordinate.

Now we have a very natural way to compose rules. Let us denote a **rule** at Processor i looking back at Processor j by $R_{i,j}$ (we use a capital letter since it's a matrix) and a **value** at Processor i by v_i. Then the following hold:

- $v_j \cdot v_i = R_{i,j}$.
- $R_{i,k} = R_{i,j} \cdot R_{j,k}$

2.2.2 Fibonacci Example

Let's illustrate with the Fibonacci recurrence, $a_i = a_{i-1} + a_{i-2}$. Here, $c_1 = 1, c_2 = 1, c_3 = d = 0$. If a processor has a value, then we store that value in the first row of the rule matrix. This saves us having to have separate data structures at each processor for the rule and the value.

Processor 0 Processor 1 Processor 2 Processor 3

$$\begin{pmatrix} 3 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Initialized rules: Processor 0 has a value, Processors 1, 2 and 3 have rules.

$$\begin{pmatrix} 3 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 5 & 3 & 2 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

After one step, Processors 0 and 1 have values, Processors 2 and 3 have rules.
Matrices for Processors 1, 2 and 3 were updated by multiplying each by the matrix to its left.

$$\begin{pmatrix} 3 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 5 & 3 & 2 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 8 & 5 & 3 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 13 & 8 & 5 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Now all processors have values.
Matrices for Processors 2 and 3 were updated by multiplying each by the matrix *two* to its left.

Notice how as in the case of binary addition, each processor was interested in computing only one bit, namely its bit in the sum, but our data structure contained many more values. We have a similar situation here, where each processor wants to compute only a single number, but its data structure has sixteen entries.

2.2.3 Finding a Single Value and Processor-specific Recurrences

We end this section by exploring the second type of question we'd like to answer about recurrences: How can we find the nth term? We may extract from our lookback idea a simple solution. Note that the nth value v_n can be computed by multiplying n copies of the **rule** matrix R, and then multiplying by the **value** vector v_0. That is, $v_n = v_0 \cdot R^n$. To find R^n rapidly, we may compute the sequence $R, R^2, R^4, R^8, \ldots, R^p$, where p is the largest power of 2 not exceeding n. We can find each term in this sequence by repeatedly squaring the previous term, taking about $\log_2 n$ steps to create the whole sequence. Then we write n as a sum of powers of 2 (which is

equivalent to writing n in binary) and multiply together the corresponding matrices. For example, $73 = 2^0 + 2^3 + 2^6$, so $R^{73} = R^{2^0} \cdot R^{2^3} \cdot R^{2^6}$.

We therefore have an algorithm for computing the nth term in a recurrence in $O(\log n)$ steps.

Note that this method would not work, for example, if the recurrence were of the form:

$$a_i = a_{i-1} + \frac{2}{i} \cdot a_{i-3} - i$$

In this case, the matrix R is not constant, which is required for the method just given. Our search algorithm described above would work just fine, however, since each processor makes its own copy of the rule for looking back, which is just some 4×4 matrix. Our composition functions would not need to be changed at all.

Chapter 3

A Parallel Addition Applet

In the parallel binary addition algorithm just discussed we had a processor compute each node of the sum by doubling its lookback on each step. We have included with this module an applet (based on an AgentSheets model) that shows another way of increasing lookback that is more hierarchical in nature, and perhaps better-suited to situations where there are more processors than can comfortably fit within a single shared-memory environment, but are instead distributed across separate compute nodes.

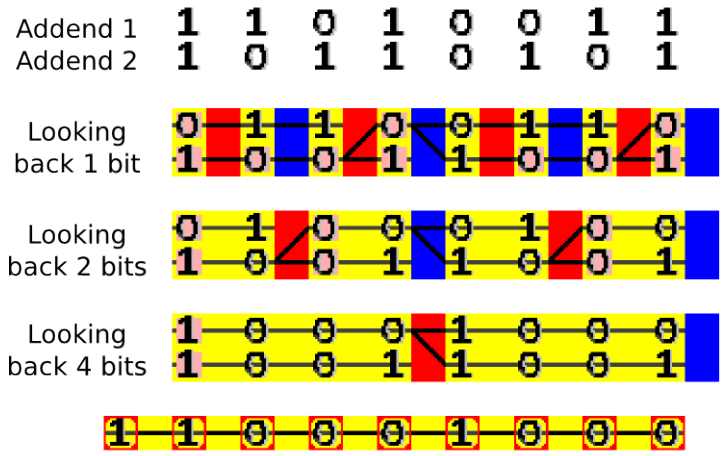

Figure 3.1: This module includes an applet showing 8-, 16- and 32-bit parallel addition.

We use the same data structure as that shown in Figure 2.2. The `sumIfNoCarry` and `sumIfCarry` bits are shown as `0`s and `1`s, but the `carryIfNoCarry` and `carryIfCarry` bits are not shown explicitly. Instead, we use shading and "wires" to depict whether there is a carry in each case. We use red shading behind each bit if that case produces a carry, and to the left of the bit pairs are "wires" that depict whether or not the processor would produce a carry in each case. Thus if a bit does not produce a carry, its wire would lead to the top row in the processor to its left, and if it does produce a carry, then its wire leads to the bottom row in the processor to its left.

Figure 3.1 shows an eight-bit example. (The applet also has 16- and 32-bit examples, and lets you set your own bits, or random bits, to add.) The top two (uncolored) rows show the two addends. After the first step, the `sumIfNoCarry` and `sumIfCarry` bits for each processor are shown in the top yellow row (with blue and red dividers) with the carry bits shown as backgrounds as described above. Every processor is looking back distance one at this point.

> As an interesting side note, we observe that at this point it is possible for the user to read off the sum visually, by following the wires starting at the top-right wire, and reading right-to-left. In fact, we have constant-time parallel addition at this point if the CPU has a "follow wire and write bits" instruction that runs in a single clock step.

Now to generate any row from the row above we apply the following rule: *For each yellow block of processors that has a red border on its right, have each processor look back to the processor just across that red border, and update its own sum structure based on what it sees there in the* `carryIfNoCarry` *and* `carryIfCarry` *bits.* The processors in a yellow block bordered on the right by a blue border do not update on that step. Thus on each step, only half the processors update their data structures, and within a yellow block, the processors are all looking back different distances. And the right-most yellow block contains the processors that know their values.

To get the answer for the addition, we continue until all processors are in the "right-most yellow block" and then read off the top row of bits. We read the top row because the sum starts with no carry from any bits to the right of the addends. Only this top row is shown at the bottom of the applet.

Chapter 4

Implementing in CUDA

We will use `float`s for all of our numbers in the CUDA implementation. These are 32-bit numbers, and are primitive data types on devices of all compute capabilities. `double`s are native only on devices of compute capability 1.3 and higher, so we won't use them here. (See section F.1 in [5] for more about compute capabilities and double-precision.)

4.1 Can we Have a Million Processors?

As we mentioned earlier, we will plan and program as if we had one processor per term in the sequence. But if we want to generate a million terms, even CUDA falls short of having a million processors. Fortunately, we may think in terms of *threads* when we program in CUDA, and CUDA allows us to have 2^{41} threads, theoretically speaking. These threads don't run all at the same time, and the blocks of threads must be able to run in any order — but otherwise we may program *as if* we had this many processors.

4.2 How Many Threads per Block?

On devices of compute capability 1.x there is 16kB of shared memory per block. Recall that "shared memory" is the fast (compared to global) memory that can be shared among the threads in a single block. Each of our threads needs 16 floats for its

```
> nvcc -Xptxas -v -DthreadsPerBlock=128 recursion.cu
ptxas info : Compiling entry function
             '_Z20copyHeadDataFromTempP4RuleS0_i' for 'sm_10'
ptxas info : Used 4 registers, 12+16 bytes smem,
             32 bytes cmem[0], 8 bytes cmem[1]
...                  ...
ptxas info : Compiling entry function
             '_Z15initializeRulesP4Rule' for 'sm_10'
ptxas info : Used 20 registers, 64+0 bytes lmem,
             8196+16 bytes smem, 32 bytes cmem[0], 16 bytes cmem[1]
```

Figure 4.1: Compiling with the `-Xptxas -v` flags shows memory usage.

rule, which corresponds to 64 bytes. 16kB/64 = 256. Thus each block may have at most 256 threads. (Note that "512" is imposed by CUDA as the maximum number of threads per block for any program running on devices of compute capability 1.x. It's "1024" for devices of compute capability 2.x. Again, see section F.1 in [5].) Unfortunately, the system will also make use of some of our shared memory, as we'll see below, so we'll have to use fewer than 256 threads per block.

There is also a limit on the number of (32-bit) registers available. Devices of compute capability 1.0 and 1.1 have 8k per multiprocessor (= 32kB), devices of compute capability 1.2 and 1.3 have 16k (= 64kB), and devices of compute capability 2.x have 32k (= 128kB). It is somewhat difficult to predict exact register usage, but it is possible to discover how the compiler has allocated registers, by compiling with the `-Xptxas -v` option. Figure 4.1 shows the output from a compiling run of the program `recursion.cu` included with this module:

The program has five kernel functions (declared `__global__` in the source). For brevity, the output for only two is shown in Figure 4.1. The output produced by these flags is not very well-documented, but what little there is can be found in The CUDA Compiler Driver NVCC [2], page 28. The third "`ptxas`" line (split into two lines for the figure) shows the kernel function being compiled, `initializeRules`, and the next line shows memory usage. We see that each thread will use 20 registers, 64 bytes of local memory, 8196+16 bytes of shared memory, 32 bytes of program-declared (`__constant__`) constant memory and 16 bytes of compiler-generated constant memory.

Note on "smem" and "lmem": Some documentation and forum postings seem to suggest that "8196+16" means that the total amount of shared memory used is 8196 bytes, *of which* 16 bytes is used by the system. A fair bit of experimentation, though, leads me to believe that the amount of shared memory used is in fact the sum of those values, or 8212 bytes. This is supported by the "12+16" value for shared memory usage in the first line ($12 < 16$, so the "12" can't include the "16") and by experiments where I allocated more and more shared memory, until I had exceeded the amount available, and got an error only and exactly when the *sum* of the two values exceeded the available amount.

We are interested in the register usage. Among the five kernel functions, `initializeRules` uses the greatest number of registers: 20. On a device of compute capability 1.1 (such as my old MacBook Pro) where there are 8192 registers available, we'd be able to run at most $8192/20 = 409$ threads per block. This is well above the limit imposed by shared memory usage, so register usage won't be a factor here.

Let's look again at shared memory usage. We see that `initializeRules` used 8196+16 bytes of shared memory when we use 128 threads per block, as indicated by the commandline flag `-DthreadsPerBlock=128`. Each thread allocates its own 4×4 array of floats, which uses 64 bytes. So we would expect to see $128 \times 64 = 8192$ bytes used, plus another four bytes for the function parameter, which is a pointer. Indeed we do see 8196 bytes, plus another 16 that the system is using. In general we can see that with T threads per block, we'll use $64T + 20$ bytes of shared memory. If we solve $64T + 20 < 16384$ for T, we find that we may have up to 255 threads per block for devices of compute capability 1.x.

4.3 Dealing with Block Independence

Since we want to search through not hundreds, but hundreds of thousands of values of the recurrence, we will need to generate many blocks of threads. The most natural configuration for our grid would be a single row containing all the blocks we want to create. CUDA imposes a limit of 65536 on the dimensions of a grid. So if we wanted all of our threads to lie in a one-dimensional grid, we could have no more than 16,711,680 threads, with one sequence element per thread. Fewer if we use (for some reason) less threads per block.

We want blocks to do as much work as possible within their shared memory, and

we need to assure that the blocks of threads can be scheduled in any order without sacrificing correctness. Our strategy is to use a two-layered approach. In the lower layer we have all the threads within the blocks, and in the higher layer we have just the *heads* of the blocks, that is, the first thread in each block. We store the rules / values for each processor (thread) on the device in an array called `devRules[]`. Here is the outline of our implementation, with the relevant functions from `recursion.cu`:

1. Initialize the `devRules[]` array so that `devRules[0]` has a value, and all other entries have rules. (`initializeRules`)
2. Threads within a block compose their rules by doubling their lookback until every thread is looking back to the head thread in the block. Threads in the first block will all have values at this point. Threads in all the other blocks will have rules. (`propagate`)
3. The head threads of all blocks (except the first) compose their rules with the last thread of the previous block, so that the head of every block (except the first) is now looking back to the head of the previous block. (`linkBlockHeads`)
4. Now we have just the heads of the blocks double their lookbacks until every head has computed a value. (`doubleHeadLookback, copyHeadDataFromTemp`)
5. Within each block we propagate the value at the head thread to all the other threads in that block via lookback doubling. (`propagateAllBlocks`)

Steps 1, 2 and 3 are each called once. Step 4 is called ceil((log$_2$(number of blocks)) times, and step 5 is called once. At the end of step 5, every thread has computed its value, and the values are stored in the `devRules[]` array. In Step 1, each thread initializes itself independent of all the other threads, and Step 2 is a per-block process, with no inter-block dependence, so there is no need to worry about the order in which blocks are executed in those steps. Step 3 does have inter-block dependence, but the value that block i needs from block $i-1$ doesn't change during the execution of Step 3, so again, there is no problem with the order in which blocks run. And Step 5, like Step 2, is a per-block process.

Step 4, however, could definitely cause trouble if we're not careful. For example, if the head from Block 8 completes all of its lookback-doubling before Block 4 computes its lookback, then Block 8 could be using an incorrect rule when it reads Block 4's rule. We achieve the needed synchronization by having a variable in `main()` called `lookback` which starts at 1, and doubles until it exceeds or equals the number of blocks. For each value of `lookback`, `main()` runs the `doubleHeadLookback()` kernel with `lookback` as a parameter. This kernel will com-

plete before it is run again with the new value of `lookback`. In this way, after every run of the `doubleHeadLookback()` kernel, the head threads are all looking back the same, predictable distance as one another.

4.4 Implementation Miscellany

All the code for this algorithm is contained in the file `recursion.cu`.

There are three `__device__` functions. These are functions that run on the device, and which are called by a kernel or other device functions. The functions `setValue()` and `updateRule()` are fairly self-explanatory, and simply carry out the algorithm described in the text. The `propagate` method is a bit more interesting. It implements Step 2, and is called at the end of Step 1, in the `initializeRules()` function. Notice that this function declares a local, two-dimensional array `tmp[4][4]` used for storing temporary copies of the thread's `rule[4][4]`. If we stored these in shared memory, then we'd have to have fewer threads per block. Fortunately we have plenty of room in the registers, so we store them there by simply declaring the `tmp` array to be local to the thread.

The two kernel functions `linkBlockHeads()` and `doubleHeadLookback()` each run half the number of threads, and process half the number of rules, per block as the other kernel functions. This is seen in the statements: `H = threadsPerBlock / 2`, where `H` is the actual number of threads per block in those kernels. The reason is that in each case, we need in shared memory *two* copies of a `Rule` data structure for each thread.

Finally, we note that for maximum flexibility we have allowed for more than 65536 blocks of threads by not assuming that our blocks all comprise a single row of the grid. (For more on blocks and grids, see Chapter 2 of the CUDA C Programming Guide [5]). We may thus have (theoretically) $255 \times 65536 \times 65536$, just over a trillion threads.

Finally, we note that the explorations given in this paper regarding the Fibonacci numbers actually run more *slowly* than the investigations would run on a single CPU with well-written code. This is because the computation we are performing on each Fibonacci number is very minimal, while the overhead of computing the value of each entry is significant, as it requires the multiplication of 4×4 matrices, as well as the many reads and writes to and from global memory. For example, if we were to do something compute-intensive with each entry (more than simply compute it) then

the gains from parallelism would make the overhead time less significant. Also, as block size increases, the number of accesses needed to global memory will decrease proportionally; and as the number of CUDA cores increases, the running time for propagating head values to the blocks will decrease proportionally as well.

Chapter 5

Lab Explorations

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 [5].) Repeat the calculation if instead of 8 bytes per thread we need 48 bytes per thread.

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.

```
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 [5] 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.

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".

(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?

(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?

(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.

(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?

(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?

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 [4] 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 5.1 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.

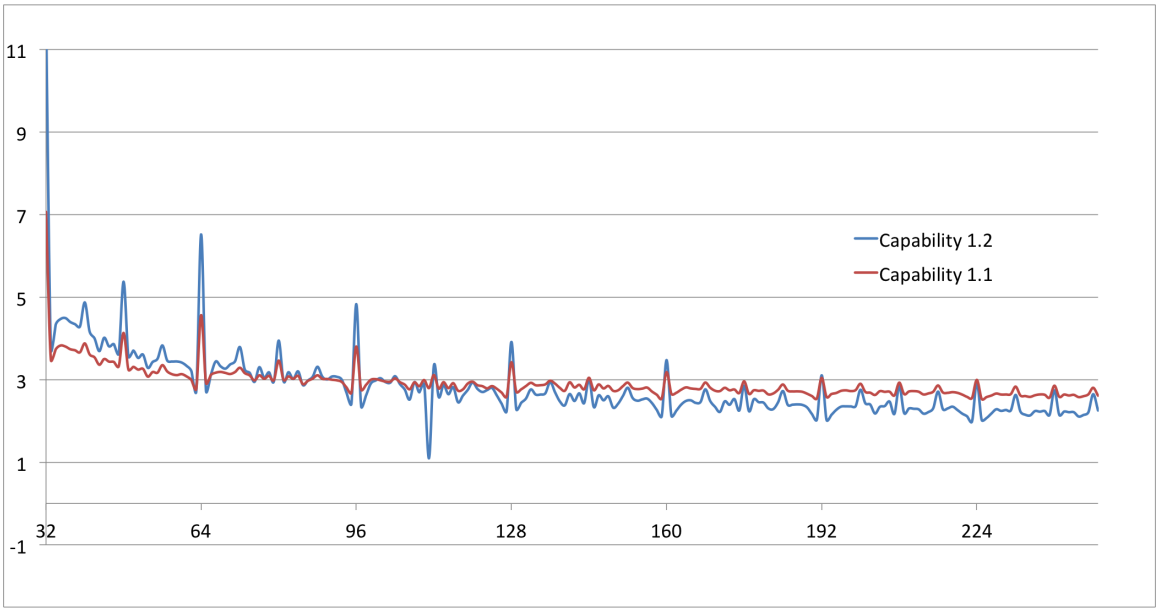

Figure 5.1: Run times in seconds vs. number of threads per block.

Chapter 6

The Code

6.1 Head Matter

```
#include <stdio.h>
#include <cuda.h>
#include <math.h>
#include <stdint.h>

typedef struct{
  float a[4][4];
} Rule;

#ifndef threadsPerBlock
#define threadsPerBlock 128
#endif

#ifndef NUMELTS
#define NUMELTS 5000
#endif

/*
 * This array defines the recursion rule.
 * a_n = rec[0]*a_{n-1} + rec[1]*a_{n-2} + rec[2]*a_{n-3} + rec[3]
 */
__constant__ float rec[4];
```

```
/*
 * This array defines the initial values
 * a[i] = init[i] for 0 <= i <= 3
 */
__constant__ float init[4];
```

6.2 Device Functions

```
/**************************************
 * replaces the rule "here" with the product of the
 * rules (4x4 matrices) found "here" and "there".
 * The "there" matrix is not changed
 *
 * Note that the caller needs to make sure there is no
 * unwanted dependency upon the order in which the blocks
 * of the grid call this function.
 **************************************/
__device__ void updateRule(Rule* rule, int here, int there){
  float tmp[4][4];
  if(there >= 0){
    for(int i = 0; i < 4; i++)
      for(int j = 0; j < 4; j++){
tmp[i][j] = 0;
for(int k = 0; k < 4; k++)
  tmp[i][j] += rule[there].a[i][k] * rule[here].a[k][j];
      }
  }

  // Write from registers back to shared memory
  for(int i = 0; i < 4; i++)
    for(int j = 0; j < 4; j++)
      rule[here].a[i][j] = tmp[i][j];
}


/**************************************
 * Replaces the "here" rule with a "here" value
 * instead. A "value" is just a row, the first row,
```

```
 * holding the value of a_n, a_{n-1}, a_{n-2} and a_{n-3}
 * for the rule at index n.
 ****************************************/
__device__ void setValue(Rule* rule, int here, int there){
  int i, k;
  for(i = 0; i < 4; i++){
    float sum = 0;
    for(k = 0; k < 4; k++){
      sum += rule[here].a[i][k] * rule[there].a[k][0];
    }
    rule[here].a[i][0] = sum;
  }
}


/***********************************************
 * This routine propagates the rule or value at the
 * head to rules or values, resp., at the other
 * threads in the block
 ***********************************************/
__device__ void propagate(Rule* rule){
  float tmp[4][4];
  int lookback = 1;
  while(lookback < threadsPerBlock){
    if(threadIdx.x < lookback + 1)
      return;
    int here = threadIdx.x;
    int there = here - lookback;
    if(here < 0)
      here = 0;

    for(int i = 0; i < 4; i++)
      for(int j = 0; j < 4; j++){
tmp[i][j] = 0;
for(int k = 0; k < 4; k++)
  tmp[i][j] += rule[there].a[i][k] * rule[here].a[k][j];
      }

    syncthreads();

    if(threadIdx.x >= lookback)
```

```
      for(int i = 0; i < 4; i++)
for(int j = 0; j < 4; j++)
  rule[here].a[i][j] = tmp[i][j];

    lookback = lookback << 1;
  }
}
```

6.3 Kernel Functions

```
/*********************************************
 * Kernel - Initialize rules
 *
 * Fills the section of memory pointed to by devRules
 * with values. So that the nth devRule contains
 * a_n, a_{n-1}, a_{n-2} and a_{n-3}
 *********************************************/
__global__ void initializeRules(Rule* devRules){
  // set up some variables
  int idx = (gridDim.x * blockIdx.y + blockIdx.x) * blockDim.x
             + threadIdx.x;
  if(idx >= NUMELTS)
    return;

  __shared__ Rule rule[threadsPerBlock];

  // Initialize the value of the very first element
  if(idx == 0){
    for(int i = 0; i < 3; i++){
      rule[idx].a[0][2-i] = init[i];
      for(int j = 1; j < 4; j++)
rule[idx].a[j][i] = 0;
    }
    rule[idx].a[0][3] = init[3];
    for(int j = 1; j < 4; j++)
      rule[idx].a[j][3] = 0;
  }
  // Initialize the rules for the rest of the elements
  else{
```

```
    for(int i = 0; i < 4; i++){
      rule[threadIdx.x].a[i][0] = rec[i];
      for(int j = 1; j < 4; j++){
rule[threadIdx.x].a[i][j] = 0;
      }
    }
    // Set up a custom value for "+d" in the line below
    rule[threadIdx.x].a[3][0] = (float)1.0/idx;
    rule[threadIdx.x].a[0][1] = 1;
    rule[threadIdx.x].a[1][2] = 1;
    rule[threadIdx.x].a[3][3] = 1;
  }
  syncthreads();

  // Have all threads in this block obtain the rule or value from
  // their head thread
  propagate(rule);
  // Copy our rule or value to the global memory
  memcpy(&devRules[idx], &rule[threadIdx.x], sizeof(Rule));
}


/*************************************
 * Kernel  -  linkBlockHeads
 *
 * This kernel gives the head of each block the rule
 * for updating from the head of the previous block
 *
 * This is called only once, just after initialization, to link the blocks.
 *
 * Note that 'threadsPerBlock' is not the number
 * of threads per block in this kernel. This kernel
 * actually has NUMELTS = threadsPerBlock/2 threads per block.
 *************************************/
__global__ void linkBlockHeads(Rule* devRules){

  int idx = (gridDim.x * blockIdx.y + blockIdx.x) * blockDim.x + threadIdx.x;
  if(idx * threadsPerBlock >= NUMELTS)
    return;

  int H = threadsPerBlock/2;
```

```
  int myHere = 2 * (idx % H);
  __shared__ Rule rule[threadsPerBlock];


  // Copy the headrule and the tail of the previous block into
  // adjacent spaces in rule
  memcpy(&rule[myHere], &devRules[idx * threadsPerBlock - 1],
         2*sizeof(Rule));
  if(idx != 0){ /* Not the first block, which needs no link */
    updateRule(rule, myHere+1, myHere);
  }

  // Write the updated rule back to memory
  memcpy(&devRules[idx * threadsPerBlock], &rule[myHere + 1],
         sizeof(Rule));
}



/*************************************
 * Kernel  -  propagateAllBlocks
 *
 * This kernel propagates the rule or value at the head of each block to
 * the other threads in its block.
 *
 * Before the call, each thread's rule is already looking back to the head thread.
 *************************************/
__global__ void propagateAllBlocks(Rule* devRules){
  int idx = (gridDim.x * blockIdx.y + blockIdx.x) * blockDim.x
            + threadIdx.x;
  if(idx >= NUMELTS)
    return;

  __shared__ Rule rule[threadsPerBlock];

  rule[threadIdx.x] = devRules[idx];
  syncthreads();
  if(threadIdx.x != 0){
    updateRule(rule, threadIdx.x, 0);
    devRules[idx] = rule[threadIdx.x];
  }
```

```
}


/************************************
 * Kernel  -  doubleHeadLookback
 *
 * This kernel composes the rule at the head of the block with the
 * rule at the head of the block "lookback"-many blocks previous,
 * and saves the composition in temporary storage.
 *
 * This is called logarithmically many times.
 ************************************/
__global__ void doubleHeadLookback(Rule* devRules, Rule* headRulesTemp,
                                   int lookback){
  int idx = (gridDim.x * blockIdx.y + blockIdx.x) * blockDim.x
            + threadIdx.x;
  if(idx * threadsPerBlock >= NUMELTS)
    return;
  //  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int H = threadsPerBlock/2;
  int myHere = 2 * (idx % H);
  __shared__ Rule rule[threadsPerBlock];

  // Return if our blocks already have values
  if(idx < lookback)
    return;

  // We will update. Copy the two rules into shared memory
  memcpy(&rule[myHere + 1], &devRules[idx * threadsPerBlock],
         sizeof(Rule));
  memcpy(&rule[myHere], &devRules[(idx-lookback) * threadsPerBlock],
         sizeof(Rule));
  syncthreads();

  updateRule(rule, myHere + 1, myHere);

  // Save this new rule or value to our temporary head memory
  memcpy(&headRulesTemp[idx], &rule[myHere + 1], sizeof(Rule));
}
```

```
/*****************************************************
 * Kernel  -  Simple kernel to copy the temporary copies of the
 * head rules back to the device array containing all rules.
 *
 * called logarithmically many times, once after each head
 * lookback doubling.
 *
 *****************************************************/
__global__ void copyHeadDataFromTemp(Rule* devRules, Rule* headRulesTemp,
                                     int lookback){
  int idx = (gridDim.x * blockIdx.y + blockIdx.x) * blockDim.x
           + threadIdx.x;
  //  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx < lookback || idx * threadsPerBlock >= NUMELTS)
    return;
  memcpy(&devRules[idx * threadsPerBlock], &headRulesTemp[idx],
         sizeof(Rule));
}
```

6.4 Host Functions

```
/****************************************
 * For printing the arrays
 ****************************************/
void printout(Rule* r, bool wholeArray){
  printf("Start of printout ************************************\n");
  for(int i = 0; i < NUMELTS; i++){
    if(true || i % threadsPerBlock == 0){ /* Which values should display? */
      if(!wholeArray)
printf("Value %d = %f%c", i, r[i].a[0][0], i%4==3 ? '\n' : '\t');
      else{
printf("************** Block %d ******************\n", i);
for(int j = 0; j < 4; j++){
  for(int k = 0; k < 4; k++){
    printf("%f ", r[i].a[j][k]);
  }
  printf("\n");
}
```

```
printf("\n");
      }
    }
  }
}


/*******************************************
 * main
 *******************************************/
int main(void){
  // Set up constant values to put on device's constant memory
  int blocksPerGrid = (NUMELTS + threadsPerBlock - 1) / threadsPerBlock;
  printf("threadsPerBlock = %d, blocksPerGrid = %d\n", threadsPerBlock,
         blocksPerGrid);

  // 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));

  // Set up some memory spaces for rules, One rule per entry
  Rule* returnRules = (Rule*) malloc(NUMELTS*sizeof(Rule));
  Rule* devRules;     /* Device copy of one rule per entry */
  Rule* headRulesTemp; /* space for one rule per block      */

  cudaError_t err = cudaMalloc(&devRules, NUMELTS*sizeof(Rule));
  printf("Malloc device rules: %s\n",cudaGetErrorString(err));

  err = cudaMalloc(&headRulesTemp, blocksPerGrid * sizeof(Rule));
  printf("Malloc temp head rules device: %s\n",cudaGetErrorString(err));

  int griddim = (int)ceil(sqrt(blocksPerGrid));
  dim3 dimGrid(griddim, (blocksPerGrid + griddim - 1)/griddim);
  initializeRules <<< dimGrid, threadsPerBlock >>> (devRules);
  err = cudaThreadSynchronize();
  printf("Kernel: initializeRules: %s\n",cudaGetErrorString(err));
```

```
int headThreadsPerBlock = threadsPerBlock / 2;
int headBlocksPerGrid = (blocksPerGrid + headThreadsPerBlock - 1) /
                        headThreadsPerBlock;
int headgriddim = (int)ceil(sqrt((double)headBlocksPerGrid));
dim3 headDimGrid(headgriddim, (headBlocksPerGrid + headgriddim - 1)/
                        headgriddim);
linkBlockHeads <<< headDimGrid, headThreadsPerBlock >>> (devRules);
err = cudaThreadSynchronize();
printf("Kernel: linkBlockHeads: %s\n",cudaGetErrorString(err));


// Now repeatedly double the head lookback, and propagate to the blocks
// We will do this logarithmically many times.
int lookback = 1;

while(lookback < blocksPerGrid){
  // Kernel to double our lookback distance
  doubleHeadLookback <<< headDimGrid, headThreadsPerBlock >>>
                              (devRules, headRulesTemp, lookback);
  err = cudaThreadSynchronize();
  printf("Kernel: doubleHeadLookback with lookback %d: %s\n", lookback,
         cudaGetErrorString(err));

  // Kernel to store temporary head data back to the devRules (permanent) array
  int tmpCopyBlocksPerGrid = (blocksPerGrid + threadsPerBlock - 1) /
                               threadsPerBlock;
  int tmpgriddim = (int)ceil(sqrt(tmpCopyBlocksPerGrid));
  dim3 tmpDimGrid(tmpgriddim, (tmpCopyBlocksPerGrid + tmpgriddim - 1)/
                               tmpgriddim);
  copyHeadDataFromTemp <<< tmpDimGrid, threadsPerBlock >>>
                              (devRules, headRulesTemp, lookback);
  err = cudaThreadSynchronize();
  printf("Kernel: tmpCopyBlocksPerGrid with %d blocks and
                     %d threads per block %s\n",
 tmpCopyBlocksPerGrid, threadsPerBlock, cudaGetErrorString(err));

  // Double the lookback
  lookback = lookback << 1;
}

propagateAllBlocks <<< dimGrid, threadsPerBlock >>> (devRules);
```

```
  err = cudaThreadSynchronize();
  printf("Kernel: propagateAllBlocks with lookback %d: %s\n",lookback,
         cudaGetErrorString(err));

  err = cudaMemcpy(returnRules, devRules, NUMELTS*sizeof(Rule),
                   cudaMemcpyDeviceToHost);
  printf("Post-kernel copy memory off of device: %s\n",
         cudaGetErrorString(err));

  printout(returnRules, false);

  printf("Done\n");
}
```

Bibliography

[1] S. Agaian. Generalized fibonacci numbers and applications. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 3484 –3488, oct. 2009.

[2] The NVIDIA Corporation. *The CUDA Compiler Driver NVCC*. NVIDIA Corporation, 2008.

[3] The NVIDIA Corporation. *The CUDA API Reference Manual, v4.0*. NVIDIA Corporation, 2011.

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

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

[6] Jishe Feng. More identities on the Tribonacci numbers. *Ars Combin.*, 100:73–78, 2011.

[7] David E. Ferguson. Fibonaccian searching. *Commun. ACM*, 3:648–, December 1960.

[8] P.D. Shipman, Z. Sun, M. Pennybacker, and A.C. Newell. How universal are fibonacci patterns? *The European Physical Journal D - Atomic, Molecular, Optical and Plasma Physics*, 62:5–17, 2011. 10.1140/epjd/e2010-00271-8.