ECE 498 AL1 Machine Problem 4
Tiled Parallel Prefix Sum

This MP is the implementation of a tiled, threaded Parallel Prefix Sum algorithm on the G80 GPU. This algorithm is also called Scan, and will be referred to as Scan in this hand out. Scan is a useful parallel building block for many parallel algorithms, such as radix sort, quicksort, string comparison, lexical analysis, stream compaction, polynomial evaluation, solving recurrences, tree operations, and histograms.

Definition of Scan:
The all-prefix-sums operation takes a binary associative operator ⊕, and an ordered set of n elements:

[*a*0, *a*1, …, *an*-1]

and returns the ordered set:

[*a*0, (*a*0 ⊕ *a*1), …, (*a*0 ⊕ *a*1 ⊕ … ⊕ *an*-1)].

For example:
If ⊕ is addition, then the all-prefix-sums on the set [3 1 7 0 4 1 6 3]would return
[3 4 11 11 15 16 22 25]
(From Blelloch, 1990, "Prefix Sums and Their Applications")

Definition of Prescan*:*
The prescan *operation takes a binary associative operator ⊕ with identity I, and an ordered set of n elements:*

[*a*0, *a*1, …, *an*-1]

and returns the ordered set:

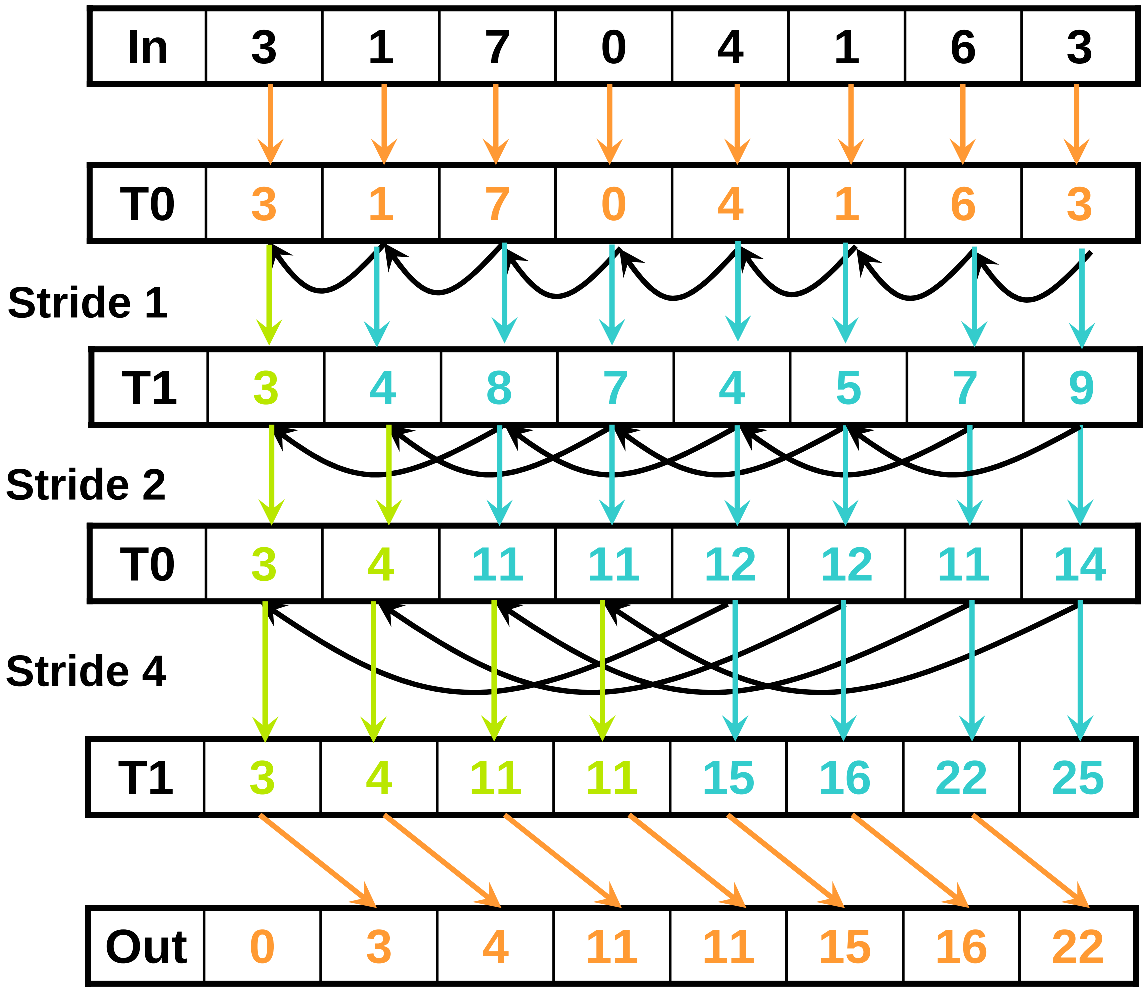
[*I*, *a*0, (*a*0 ⊕ *a*1), …, (*a*0 ⊕ *a*1 ⊕ … ⊕ *an*-2)].

For example:
if ⊕ is addition, then prescan on the set
[3 1 7 0 4 1 6 3] would return [0 3 4 11 11 15 16 22]
Scan can be trivially generated from prescan and vice-versa

Prescan on the CPU:
```
void prescan( float* scanned, float* input, int length)
{
  reference[0] = 0;
  for(int i = 1; i < length; ++i)
  {
    scanned[i] = input[i-1] + scanned[i-1];
  }
}
```

Simply add each element to the sum of the elements before it. It's trivial, but sequential.

A Naive Parallel Scan Algorithm:



However, this not work efficient!

Step complexity S(n): # of parallel iterations

Prescan executes log(n) parallel iterations

Work complexity W(n): the total # of operations

The steps do n-1, n-2, n-4,... n/2 adds each

Total adds: n * (log(n) – 1) + 1 → O(n*log(n)) work

However, to be Work Efficient:

work complexity == complexity of sequential algorithm

Sequential prescan algorithm only does *n* adds

This algorithm is NOT work efficient

A factor of log(n) hurts: 20x for 10^6 elements!

To build a work efficient scan alogrithm:

We can use *Balanced Trees*, a common parallel algorithms pattern:

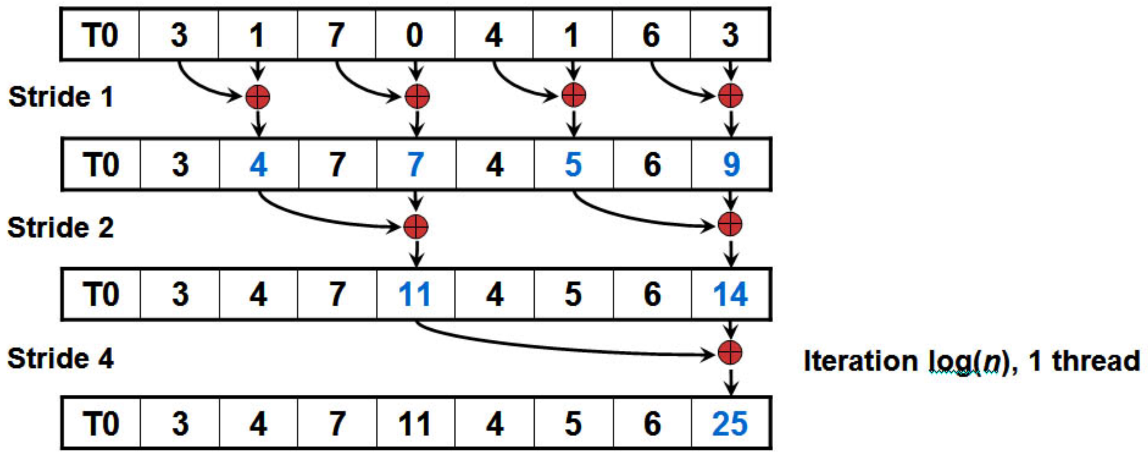
Build a balanced binary tree on the input data and sweep it to and from the root Tree is not an actual data structure, but a concept to determine what each thread does at each step.

Algorithm for prescan:

Traverse down from leaves to root building partial sums at internal nodes in the tree, where the root holds sum of all leaves.

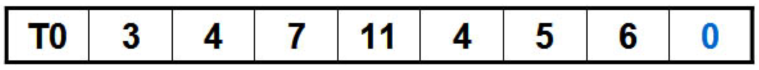
Traverse back up the tree building the scan from the partial sums

To build a sum tree:



Iterate log(n) times. Each thread adds value *stride* elements away to its own value.

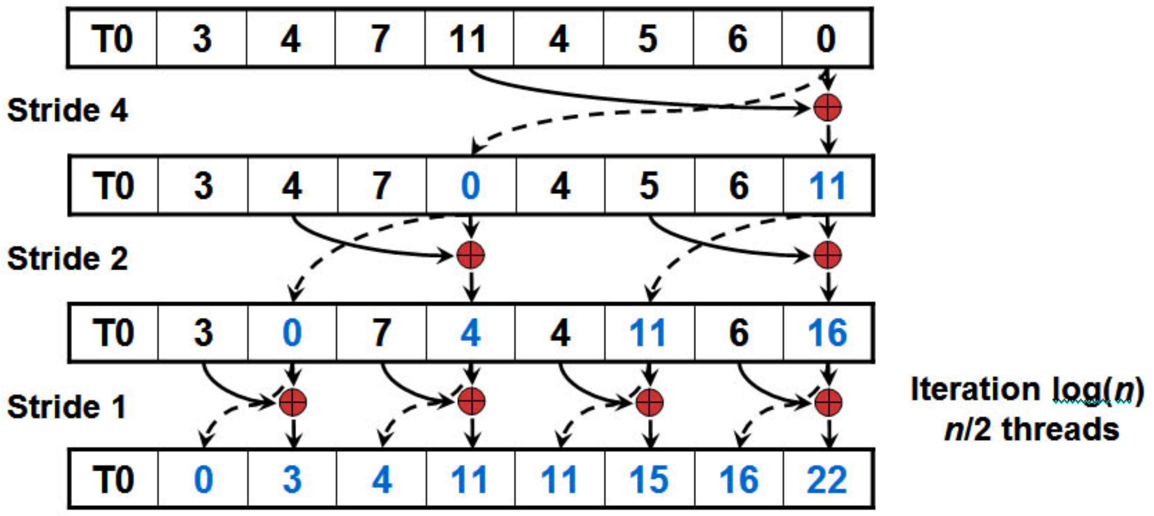
Then zero the last element:



T0	3	4	7	11	4	5	6	0

We now have an array of partial sums. Since this is a prescan,
set the last element to zero. It will propagate back to the first element.

Build prescan from partial sums:



We now have a completed prescan that we can write out to device memory.

Total steps: 2 * log(*n*).
Total work: 2 * (*n*-1) adds = *O*(*n*)   **Work Efficient!**

Goals for this MP:

1) Blocked implementation of prescan, so that we can perform prescan on more than 512 elements.
2) Work efficient algorithm, to get O(n) performance, instead of O(n*log(n)).
   Note: you do not need to use the algorithm described in this hand out exactly.
3) Handling for non-power-of-2 array sizes.
4) Implementation that minimizes shared memory bank conflicts.

Instructions:

1) Unzip MP4-tiled_scan.zip into <Proj_Dir>/NVIDIA_CUDA_SDK_1.0/projects

2) Edit the source files *scan_largearrayl.cu* to complete the necessary kernel functions.

   Use a tiled implementation so that computation can be performed on more than 512 elements on CUDA. Compile using the provided solution file with Visual Studio, or the provide Makefile file in a Linux environment.

   NOTE:
   The original nVidia Linux Makefile does not make a complete dependency check on included files. If you make changes to the kernel code in your _kernel.cu file, you would need to "make clean", and then "make" to rebuild the release build directory. Alternatively, use "make clean dbg=1", then "make dbg=1" to clean up and rebuild the debug build directory.

3) The modes of operation for the application are as follows:

   No arguments: Randomly generate input data and compare against the host's result.
   One argument: Randomly generate input data and write the result to file name specified by first argument
   Two arguments: Read the first argument which indicate the size of the array, randomly generate input data and write the input data to the second argument. (for generating random input data)
   Three arguments: Read the first file which indicate the size of the array, then input data from the file name specified by 2nd argument and write the SCAN output to file name specified by the 3rd argument.

   Note that if you wish to use the output of one run of the application as an input, you must delete the first line in the output file, which displays the accuracy of the values within the file. The value is not relevant for this application.

4) Report:
   Please use the following steps for your performance analysis and report:
   1. Near the top of scan_largearray.cu, set #define DEFAULT_NUM_ELEMENTS to 16000000 (that is, 16 million), or 16777216 if you cannot get non-powerof-2 array sizes to work. Set #define MAX_RAND to 3. We want to keep the numbers small so the 32-bit floating point units don't get saturated.
   2. Copy and paste the performance results when run without arguments, include the Host CPU and G80 CUDA processing times, the Speedup, and report whether this is the debug or release build. We will rebuild your code to confirm results.

3. Describe which of the 4 goals for this MP (mentioned above) were achieved. How did you achieve them? Did you use any clever tricks to optimize for best performance? Please briefly discuss them.
4. What is the measured floating-point computation rate (FLOPS) for the CPU (single threaded gold code) and GPU kernels on this application? How do they compare to the theoretical maximum computation limits?
5. Given that the G80 has demonstrated running at least up to ~380 GFLOPS of computation, what are some reasons that your kernel isn't reaching that limit?

5) To submit your solution, send a zipped file as an email attachment to ece498alTA@gmail.com. The subject line of the email should read "ECE498AL MP4 submission". Make sure that your name and NetID are included in the body of the email, especially if you are sending it from a non-uiuc account.

The .zip file should contain the MP4-tiled_scan folder as provided, with all the changes and additions you have made to the source code. Include your report in a text file, Word document (no Office 2007 .docx format please), or PDF file.

Grading:

Your submission will be graded on the following parameters.

Demo/knowledge: 25%
- Produces correct result output file for command-line inputs

Functionality: 40%
- Correct identification and expression of parallelism in the problem. (10%)
- Blocked Implementation (10%)
- Work Efficient Algorithm (10%)
- Handling of non-power-of-2 array sizes (5%)
- Implementation minimizes shared memory bank conflicts (5%)

Report: 35%
- Answer to provided question

Due date: Wednesday, Oct 3rd at 11:59pm.