ECE 408 Exam 2, Fall 2017

December 12th, 2017

- You are allowed one 8.0x11.5 cheat sheet with notes on both sides. The minimal font size for your text on the cheat sheet should be 8pts.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 150 minutes to complete. To allow for any unforeseen difficulties, we will consider giving everyone up to 180 minutes.
- This exam is based on lectures, textbook chapters, as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered.
- You can write down the reasoning behind your answers for possible partial credit.

Good luck!

Name: ________________________

Netid: ________________________

UIN: ________________________

Question 1: ____________________

Question 2: ____________________

Question 3: ____________________

Question 4: ____________________

Question 5: ____________________

Question 6: ____________________

Name: ____

Question 1 (30 points, 40 minutes): multiple-choice and short-answer questions. If you get more than 30 points by answering all questions (1-10), your score will saturate at 30 points. The bonus question is extra.

For multiple-choice questions, give a concise explanation for your answer for possible partial credit. Answer each of the short-answer questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

1. (3 points) For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 2048 elements, which of the following values is the closest to the total number of useful floating-point add operations performed in both the reduction tree phase and the inverse reduction tree phase?
   (A) (1024-1)*2
   (B) (2048-1)*2 - 11
   (C) 1024*2048
   (D) 2048*11

   Answer: (B)
   Explanation: For N elements, the reduction tree performs N-1 floating-point add operations performed in the reduction tree phase and the inverse reduction performs N-1-log(N) floating-point add operations. So the total is 2*(N-1) – log(N).

2. (4 points) For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 2048 elements in each section and warp size is 32, how many warps in each block will have control divergence during the reduction tree phase iteration where stride is 64?
   (A) 0
   (B) 1
   (C) 16
   (D) 32

   Answer: (B)
   Explanation: All active threads are consecutive starting with index 0. There are a total of 1024 threads and 16 active threads in each block when the stride is 64. The active threads form half of warp zero. All other warps are completely inactive. Only warp 0 has control divergence.

3. (4 points) For a processor that supports atomic operations in L2 cache, assume that each atomic operation takes 5ns to complete in L2 cache and 500ns to complete in

DRAM. Assume that 99.9% of the atomic operations hit in L2 cache. What is the approximate throughput for atomic operations on the same global memory variable?

(A) 1/500 G atomic operations per second
(B) 1/50 G atomic operations per second
(C) 1/5 G atomic operations per second
(D) 1/13.6 G atomic operations per second

Answer: (C)

Explanation: The average latency is 5ns * 99.9% + 500ns *0.1% = 5.5ns. The average throughput is approximately 1/5 G atomic operations per second

4. (3 points) In Question 3, if a kernel performs 10 floating-point operations per atomic operation, what is the approximate maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?

(A) 1 GFLOPS
(B) 2 GFLOPS
(C) 10 GFLOPS
(D) 50 GFLOPS

Answer: (B)

Explanation: 10 floating-point operations every 5.5 ns, approximately 10/(5ns) = 2 GFLOPS

5. (3 points) Given a sparse matrix of integers with R original rows, L non-zero elements in the original row with the largest number of non-zeros, and a total of N non-zeros. How many integers are needed to represent the matrix in JDS-T? Assume that we keep track of the number of non-zeros in each original row, as we specified in the MP assignment.

(A) R+L+N
(B) 2R+L+N+1
(C) 2R+L+2N
(D) 2R+2L+N

Answer: (C)

Explanation: We need N integers for the non-zero elements, N integers for the column indices of the non-zero elements, L pointers to the beginning of each sorted column after transposition, R integers to track the length of each row, and R integers to track the original row index before permutation

6. (4 points) For a sparse matrix-vector multiplication (SpMV) with R rows, a total of N non-zero elements, and a maximal of L non-zeros in each row, how many times is each matrix element used?

(A) 1
(B) 2
(C) R
(D) L

Answer: (A)
Explanation: Each matrix element will be used only once

7. (4 points) For a sparse matrix-vector multiplication (SpMV) with R rows, C columns, a total of N non-zero elements, how many times will each vector element be used?
(A) 1
(B) N/R on average
(C) N/C on average
(D) R

Answer: (C)
Explanation: A total of N accesses will be made to the vector elements. There are C of them. So each us used N/C on average.

8. (4 points) Keven has a 320MB array that he would like to process with GPU. He measured that the execution time of the code on CPU was 0.2 seconds. He also implemented a kernel and measured that the kernel execution on the GPU with the data in the GPU memory was 0.004 seconds, a 50x speedup! However, he needs to transfer the data into the GPU memory and transfer 320MB data output data back to the host memory. His system has a PCIe Gen3 x16 interconnect. What would the real speedup be?
(A) 40x speedup
(B) 20x speedup
(C) 5x speedup
(D) 1.5x slow down

Answer: (C)
Explanation: The data copy will take (320M)/(16*1000M) = 0.02 sec to copy the input data in and another 0.02 sec to copy the output data back. Thus the total speedup is 0.2/(0.004+0.02+0.02) ≈ 0.2/(0.04) = 5x speedup

9. (2 points) For the following host code sequence:

```
1) cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),.., stream0);
2) cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),.., stream0);
3) cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),.., stream1);
4) cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),.., stream1);
5) cudaMemcpyAsync(d_A2, h_A+i+2*SegSize, SegSize*sizeof(float),.., stream2);
6) cudaMemcpyAsync(h_C+i+2*SegSize, d_C2, SegSize*sizeof(float),.., stream2);
```

Which of the statements could be executed in parallel on the GPU

(A) 1) and 2)
(B) 2) and 3)
(C) 1) and 3)
(D) 2) and 4)

Answer: (B)
Explanation: API operations in different streams can be executed in parallel. However, there is only one PCIe copy engine in each direction. So, only 2) and 3) or go in parallel

10. (2 points) When your parallel reduction kernel generates a slightly different result than a sequential reduction function for a `float` input array, what would be the most likely reason?

(A) The hardware failed
(B) There is a missing `__syncthreads()` call
(C) Floating-point operations are not necessarily commutative nor associative
(D) CPU and GPUs have different precision for `float` numbers

Answer: The operation order has changed, and in one of the orders, one or more additions involved a very small operand and a much larger operand.

(Bonus 2 points) List the errors and typos that you reported via Piazza postings, e-mails, or in person communication with Prof. Hwu or Carl Pearson. (0.5 points for each item.)

Each one worth 1/2

Question 2 (15 points, suggested time allocation 20 minutes): This question tests your understanding of parallel histogram computation and privatization.

You are the owner of a supermarket and want to know the distribution of the price tags of your merchandise. So you decided to build a histogram with intervals of $5(e.g $0 <= histo[0] < $5, $5 <= histo[1] < $10 and so on) on a GPU. However, the shared memory can only accommodate 256 bins for each block. As a compromise, you decide to privatize the first 256 of the bins into the shared memory. Whenever the data value doesn't fall in the first 256 bins, you will have to increment the global bins. Assume that the prices are integer values and the **global histogram array is sized to accommodate the prices of all the items.** Also, assume that all global histogram elements have been initialized to zero before the kernel is launched.

(A) (3 Points) Complete the following kernel to implement the partial privatization of the histogram.

```
/* histo_kernel is launched with the following parameters

 dim3 gridDim(8);
 dim3 blockDim(256);
*/

__global__ void histo_kernel(unsigned int *prices, long size, unsigned int *histo){
  __shared__ unsigned int histo_private[256];

   // Reset histogram
   histo_private[__threadidx.x (+.5) ] = 0;
   __syncthreads();

   int i = threadIdx.x + blockIdx.x * blockDim.x;
   // stride is total number of threads
   int stride = blockDim.x * gridDim.x;

   while (i < size) {
    if ( prices[i] <___1280  (+.5) ) atomicAdd(__&(histo[prices[i] / 5]) (+.5)_, 1);
    else atomicAdd(__&(histo[prices[i] / 5]) (+.5)_, 1);
    i += stride;
   }
   __syncthreads();

    // contribute to global histogram
   atomicAdd( __&(histo[threadIdx.x]) (+.5)_ , __histo_private[threadIdx.x] (+.5)_ );
}
```

(B) (4 Points) how many **non-atomic global memory reads/writes** and **shared-memory/global-memory atomic operations** are being performed by all the threads executing your kernel if there are **4096** items in total which are all priced less than $1280?

Non-atomic Global Memory reads: 4096

Non-atomic Global Memory writes: 0

Shared-memory atomic operations: 4096

Global-memory atomic operations: 8 * 256

Explanation:

(C) (4 Points) how many **non-atomic global memory reads/writes** and **shared-memory/global-memory atomic operations** are being performed by all the threads executing your kernel if there are **4000** items in total which are all priced less than $1280?

Non-atomic Global Memory reads: 4000

Non-atomic Global Memory writes: 0

Shared-memory atomic operations: 4000

Global-memory atomic operations: 8 * 256

Explanation:

(D) (4 Points) how many **non-atomic global memory reads/writes** and **shared-memory/global-memory atomic operations** are being performed by your kernel if there are 4000 items are priced less than $1280 and 96 items are priced above $1280?

Non-atomic Global Memory reads: 4096

Non-atomic Global Memory writes: 0

Shared-memory atomic operations: 4000

Global-memory atomic operations: 8 * 256 + 96

Explanation:

Name: ______________________________________

Question 3: Parallelization (15 points, suggested time allocation 25 minutes):

Consider the **dense** matrix-vector multiplication **Ax** = **b**. The ith element of **b** is the dot product of **x** with the ith row of **A**.

A is a pointer to a $numRows \times numCols$ row-major matrix,
b is a pointer to a vector of length *numRows* with all entries initialized to 0, and
x is a pointer to a vector of length *numCols*.

The following is a CPU sequential code that needs to be parallelized:

```
void mv_cpu(float *b, const float *A, const float *x, const int numRows,
              const int numCols) {
  for (int colIdx = 0; colIdx < numCols; ++colIdx) {
    for (int rowIdx = 0; rowIdx < numRows; ++rowIdx) {
      b[rowIdx] += A[rowIdx * numCols + colIdx] * x[colIdx];
    }
  }
}
```

Your colleague proposes the following three parallelizations to take advantage of GPU parallelism:

```
/* mv_gpu_1 launched with the following parameters
 dim3 gridDim(10);
 dim3 blockDim(256);
*/
__global__ void mv_gpu_1(float *b, const float *A, const float *x,
                         const int numRows, const int numCols) {
  const int rowStart = blockIdx.x * blockDim.x + threadIdx.x;

  for (int r = rowStart; r < numRows; r += gridDim.x * blockDim.x) {
     float dot = 0;
     for (int c = 0; c < numCols; ++c) {
       dot += A[r * numCols + c] * x[c];
     }
     b[r] = dot;
  }
}
```

```
/* mv_gpu_2 launched with the following parameters
 dim3 gridDim(16);
 dim3 blockDim(256);
*/
__global__ void mv_gpu_2(float *b, const float *A, const float *x,
                         const int numRows, const int numCols) {
  int colStart = blockIdx.x * blockDim.x + threadIdx.x;

  for (int c = colStart; c < numCols; c += gridDim.x * blockDim.x) {
    const float v = x[c];
    for (int r = 0; r < numRows; ++r) {
      atomicAdd(&b[r], A[r * numCols + c] * v);
    }
  }
}
```

```
/* mv_gpu_3 launched with the following parameters
 dim3 gridDim(4, 4);
 dim3 blockDim(16,16);
*/
template <size_t BS>  // Assume that BS is 16
__global__ void mv_gpu_3(float *b, const float *A, const float *x,
                         const int numRows, const int numCols) {

  __shared__ float b_s[BS];
  const int tx = threadIdx.x;
  const int ty = threadIdx.y;
  const int colStart = blockIdx.x * blockDim.x + tx;
  const int rowStart = blockIdx.y * blockDim.y + ty;

  for (int r = rowStart; r < numRows; r += gridDim.y * blockDim.y) {
    if (ty == 0)
      b_s[tx] = 0;
    __syncthreads();

    for (int c = colStart; c < numCols; c += gridDim.x * blockDim.x) {
      atomicAdd(&b_s[ty], A[r * numCols + c] * x[c]);
      __syncthreads();
    }

    if (tx == 0)
      atomicAdd(&b[r], b_s[ty]);
    __syncthreads();
  }
}
```

For each column of the following table, choose entries from the first row of that table that apply to the kernel in question.

- The first column should contain only A-D, and the second column should contain only E-J.

Name: ____

- Each box may contain 0, 1, or more than one letter.

(Hint) It might be more efficient to focus on one of (A)-(J) and determine if it is applicable to mv_gpu_1 through mv_gpu_3 before moving to the next one.

	could be fast because	**could be slow because**
Possible choices for this column	(A) Coalesced memory accesses to A (B) numRows global memory writes (C) Many active threads if numCols is large (D) Many active threads if numRows is large	(E) contention in shared memory atomics (F) Barrier synchronization (G) numRows & numCols global memory writes (H) redundant loads from X (I) uncoalesced memory accesses to A (J) contention in global memory atomics
mv_gpu_1	(D) (B)	(I) (H)
mv_gpu_2	(A) (C)	(J) (G)
mv_gpu_3	(D) (A) (C)	(F) (E) (J) (H)

Question 4. Sparse Matrix Multiplication (10 points, suggested time allocation 25 minutes):

This question tests your knowledge of Sparse Matrix representation and operation. For your convenience, we are enclosing an example that illustrates how a dense matrix is transferred to the sparse matrix in ELL format. Recall that we take CSR, pad elements to make all rows of equal length, and transpose the padded matrix.

Name: ________________________________

Dense Matrix:

$$\begin{array}{l|cccc}
\text{Row 0} & 3 & 0 & 1 & 0 \\
\text{Row 1} & 0 & 0 & 0 & 0 \\
\text{Row 2} & 0 & 2 & 4 & 1 \\
\text{Row 3} & 1 & 0 & 0 & 1
\end{array}$$

Sparse Matrix in ELL:

data:

3	*	2	1	1	*	4	1	*	*	1	*

col_index:

0	*	1	0	2	*	2	3	*	*	3	*

num_elem = 3
num_rows = 4

(A) (4 points) In the following ELL kernel, fill in the missing indexing expressions for accessing data (input matrix), x (input vector) and y (output vector).

```
__global__ void SpMV_ELL (int num_rows, float *data, int *col_index, int num_elem, float *x, float *y) {
 int row = blockIdx.x * blockDim.x + threadIdx.x;
 if (row < num_rows) {
   float dot = 0;
   for (int i = 0; i < num_elem; i++) {
          dot += data[_____________] * x[_____________];
   }
   y[row] = dot;
   }
 }
```

row+i*num_rows

col_index[row+i*num_rows]

(B) (4 points) After we transform the matrix into ELL layout as the example shows, and launch the kernel in (A), circle one answer for each question below and **justify your choice**.

(1) Is there any control divergence (assuming num_rows is a multiple of 32)?
Answer: **Yes** or **No**

Explanation:

No. With zero-padding, all rows are of the same length. All threads iterate the same number of times in the dot product loop. Thus, control flow divergence no longer occurs in warps.

(2) Is memory access coalesced?
Answer: **Yes** or **No**

Explanation:

Yes. As all the elements are arranged into column major order, all adjacent threads are accessing adjacent memory locations which enables memory coalescing.

(C) (2 points) State what the hybrid ELL-COO format is and give a situation when hybrid ELL-COO method performs better than the ELL format.

In the situation where one or a small number of rows have an exceedingly large number of nonzero elements, the ELL format will result in excessively number of padded elements. To solve this, we can use the hybrid ELL-COO method which "take away" some of the excessive elements to reduce zero-paddings.

Question 5. Convolution Neural Network (15 points, suggested time allocation 15 minutes):

A basic convolution layer consists of filter W, input X and output Y. We want to accelerate the forward propagation of convolution layers in the training process.

W is the convolution filter weight tensor, organized a tensor W[M, C, K, K], M is the number of output feature maps, C is the number of input feature maps, K is the height and width of each filter. Tensors are stored as multi-dimensional arrays in the memory.

X is the input feature map, organized as a tensor X[B, C, H, W], where B is the number of images in one mini-batch (recall that mini-batch is used to efficiently updates gradients while keeping relatively fast convergence), H is the height of each input feature map and W is the width of each input feature map.

Y is the output feature map, organized as a tensor Y[B, M, H_out, W_out], where H_out = H-K+1 is the height of each output feature map and W_out = W-k+1 is the width of each output feature map.

(A) You first start with a naïve implementation for the first convolutional layer given that an output feature map of H_out x W_out (28 * 28) can fit in one block (so that each thread is processing one element in the output). You need to fill in the missing parts so that the convolution layer is complete.

```
    // assume that the kernel will be launched with the following configuration
dim3 gridDim(B, M, C);
dim3 blockDim(W_out, H_out, 1);

__global__ void convLayerForward_Naive(int C, int K, int W_out, int H_out,
                                       float* X, float* W, float* Y){
  int b = ________________;
  int m = ________________;
  int c = ________________;
  int h = ________________;
  int w = ________________;

  float acc = 0;
  for (int p = 0; p < K; p++) // KxK filter
    for (int q = 0; q < K; q++)
      acc += X[b, c, ________________, ________________] * W[m, c, p, q];

  Y[b, m, h, w] = acc;
}
```

Answer:

```
    // assume that the kernel will be launched with the following configuration
dim3 gridDim(B, M, C);
```

```
02: dim3 blockDim(W_out, H_out, 1);
03:
04: __global__ void convLayerForward_Naive(int C, int K, int W_out, int H_out,
                                           float* X, float* W, float* Y){
05:   int b = blockIdx.x;
06:   int m = blockIdx.y;
07:   int c = blockIdx.z;
08:   int h = threadIdx.y;
09:   int w = threadIdx.x;
10:
11:   float acc = 0;
12:   for (int p = 0; p < K; p++) // KxK filter
13:     for (int q = 0; q < K; q++)
14:       acc += X[b, c, h + p, w + q] * W[m, c, p, q];
15:
16:   Y[b, m, h, w] = acc;
17: }
```

(B) You then find out that the x, y, z dimension in the thread blocks can be mapped arbitrarily. Suppose now you change blockDim to blockDim(H_out, W_out, 1), which line(s) in the kernel of part (1) need to be changed in order to be an correct implementation?

New `gridDim`:

```
01: dim3 gridDim(B, M, C);
02: dim3 blockDim(H_out, W_out, 1);
```

Line(s) need to be changed (You may not need to fill in all the lines):

Line ________, change to: ____.
Line ________, change to: ____.
Line ________, change to: ____.
Line ________, change to: ____.
Line ________, change to: ____.

Answer:

Line 8, change to: `int h = threadIdx.x`.
Line 9, change to: `int w = threadIdx.y`.

(C) Comparing the kernel in part (2) to the kernel in part (1), would you expect the performance of kernel in part (2) to increase, decrease or stay the same? Explain why.
The performance will drop because memory coalesce is not achieved. When threadIdx.x is mapped to w consecutive addresses are accessed within a warp.

Name: ______________________________

Question 6. Scan (15 points, suggested time allocation, 25 minutes): You are a new hire at a Parallelism for Cheap Inc. The company specialises in creating parallel algorithms for machines with cheaper hardware. Your boss heard you took Applied Parallel Programming with the Wen-Mei Hwu and wants you show your expertise. Your boss wants you to make some changes to the Brent-Kung algorithm to further improve the efficiency of the hardware use.

You reason that you can further improve the execution efficiency by using fewer threads to do more work. For example, instead of using 1024 threads in each block to process 2048 elements in each section, you would like to use 64 threads in each block. To do this you break up the work at each level of the reduction and post-scan steps into parts.

At each level, all threads in the block will process one part of their section of input elements before moving onto the next part. For example, at the first level of the reduction tree, all 64 threads will process the first 128 elements (64 pairwise additions) as part 0. They will then move to process the next 128 elements as part 1. So the threads will iterate through 16 parts for the first level of the reduction tree.

(A) (5 Points) With 2048 elements in each section, about how many times more computations will each thread be doing compared to a regular Brent-Kung thread on average? Answer should be in the form 1 times, 5 times, 100 times etc. Explain for full credit.

Answer: For the new kernel, 64 threads will do the same amount of work that used to be done by 1024 threads, so each thread will be doing 16 times more work on average.

(B) (5 Points) Below is the skeleton code for implementing this strategy. The given code is very similar to your MP. Fill in the missing conditionals and indexing to make the code run as described in part A. Note that you may leave a blank empty if you feel nothing should go there. No explanation is necessary.

```
#define BLOCK_SIZE 64
#define SECTION_SIZE 2048
__global__ void scan(float *input, float *output, int len) {
  __shared__ float shared[SECTION_SIZE];
  int bx = blockIdx.x;
  int tx = threadIdx.x;
  int i = bx * SECTION_SIZE + tx;

  for(int part = 0; part < SECTION_SIZE / BLOCK_SIZE; part++) {
    if(i + part * BLOCK_SIZE < len)
      shared[tx + part * BLOCK_SIZE] = input[i + part * BLOCK_SIZE];
    else
      shared[tx + part * BLOCK_SIZE] = 0;
  }

  for(unsigned int stride = 1; stride < SECTION_SIZE; stride *= 2) {
    __syncthreads();
    for(int part = 0; part < __________________________________________; part++){
      int index = (tx+1+__________________) * (stride *2) - 1 ____________;
```

Name: ______________________________________

```
      if(index < SECTION_SIZE)
        shared[index] = shared[index] + shared[index - stride];
    }
  }

  for(unsigned int stride = (SECTION_SIZE)/4; stride > 0; stride /= 2) {
    __syncthreads();
    for(int part = 0; part < __________________________________; part++){
      int index = (tx+1+_______________) * (stride *2) - 1 __________;
      if(index + stride < SECTION_SIZE)
        shared[index + stride] = shared[index + stride] + shared[index];
    }
  }

  __syncthreads();
  for(int part = 0; part < SECTION_SIZE / BLOCK_SIZE; part++){
    if(i + part * BLOCK_SIZE < len)
      output[i + part * BLOCK_SIZE] = shared[tx + part * BLOCK_SIZE];
  }
}
```

Answer:

```
#define BLOCK_SIZE 512
#define SECTION_SIZE 4096
__global__ void scan(float *input, float *output, int len) {
  __shared__ float shared[SECTION_SIZE];
  int bx = blockIdx.x;
  int tx = threadIdx.x;
  int i = bx * blockDim.x * (SECTION_SIZE / BLOCK_SIZE) + tx;

  for(int block = 0; block < SECTION_SIZE / BLOCK_SIZE; block++) {
    if(i + block * BLOCK_SIZE < len)
      shared[tx + block * BLOCK_SIZE] = input[i + block * BLOCK_SIZE];
    else
      shared[tx + block * BLOCK_SIZE] = 0;
  }

  for(unsigned int stride = 1; stride < SECTION_SIZE; stride *= 2) {
    __syncthreads();
    for(int part = 0; part < SECTION_SIZE/(BLOCK_SIZE*2); part++){
      int index = (tx+part*BLOCK_SIZE+1)*(stride*2) - 1;
      // (tx+1)*(stride*2) + part*BLOCK_SIZE * (stride*2)- 1
      if(index < SECTION_SIZE)
        shared[index] = shared[index] + shared[index - stride];
    }
  }

  for(unsigned int stride = (SECTION_SIZE)/4; stride > 0; stride /= 2) {
    __syncthreads();
    for(int part = 0; part < SECTION_SIZE/(BLOCK_SIZE*2); part++){
      int index = (tx+part*BLOCK_SIZE+1)*(stride*2) - 1;
```

Name: ______________________________________

```
        // (tx+1)*(stride*2) + part*BLOCK_SIZE * (stride*2)- 1
        if(index + stride < SECTION_SIZE)
          shared[index + stride] = shared[index + stride] + shared[index];
      }
    }

    __syncthreads();
    for(int block = 0; block < SECTION_SIZE / BLOCK_SIZE; block++){
      if(i + block * BLOCK_SIZE < len)
        output[i + block * BLOCK_SIZE] = shared[tx + block * BLOCK_SIZE];
    }
  }
```

(C) (5 Points) Does this strategy optimally use the hardware efficiently in terms of active threads and control divergence? Explain why or why not. If it does not, how could you improve the strategy? Your answer must begin with yes or no followed by an explanation.

Answer: Yes. By separating the computations into parts, as we proceed through the reduction and post scan steps we turn off half the parts, but all of the threads will still do the same amount of work. This leads to having as many threads as possible being active at any given iteration, thus optimally using hardware efficiently. In addition, the threads that are active at any given iteration will all be adjacent minimizing control divergence.

For 2048 elements, The average number of active threads is

(64 + 64 + 64+ 64 + 64 + 32 +16 + 8 + 4 + 2 +1) + (2-1) + (4-1) + (8-1) + (16-1) + (32-1) + (64-1) + 64 + 64 + 64 + 64 which is approximately 64*10+ 2*32 + 2*32 = 768
(16 + 8 + 4 + 2 + 1) * 64 + 32 + 16 + 8 + 4 + 2 + 1 + (16 + 8 + 4 + 2 + 1) * 64 - 1*5 + 31 + 15 + 7 + 3 + 1 / ((16 + 8 + 4 + 2 + 1 + 16 + 8 + 4 + 2 + 1 + 6 + 5) * 64) which is approximately (64 * 64) / (73 * 64) = 64/73, threads are active about 88% of the time.

The average number of active threads is 768/21 =36
The average portion of threads that are active is 36/64

For regular Brent-Kung, the average number of active threads is

(1024 + 512 + … + 1 + (2-1) + (4-1) + (8-1) +... + (1024-1))/21 which is approximately
2*2048/21 = 195
(1024 + 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 + (2-1) + (4-1) + (8-1) + (16-1) + (32-1) + (64-1) + (128-1) + (256-1) + (512-1) + (1024-1)) / (21 * 1024) which is approximately (2 * 2048) / (21 * 1024) = 4/21, threads are active about 19% of the time.

So, on average, only 195/1024 = 19% of the threads are active

Name: ____

The new Brent-Kung has 4x better execution efficiency