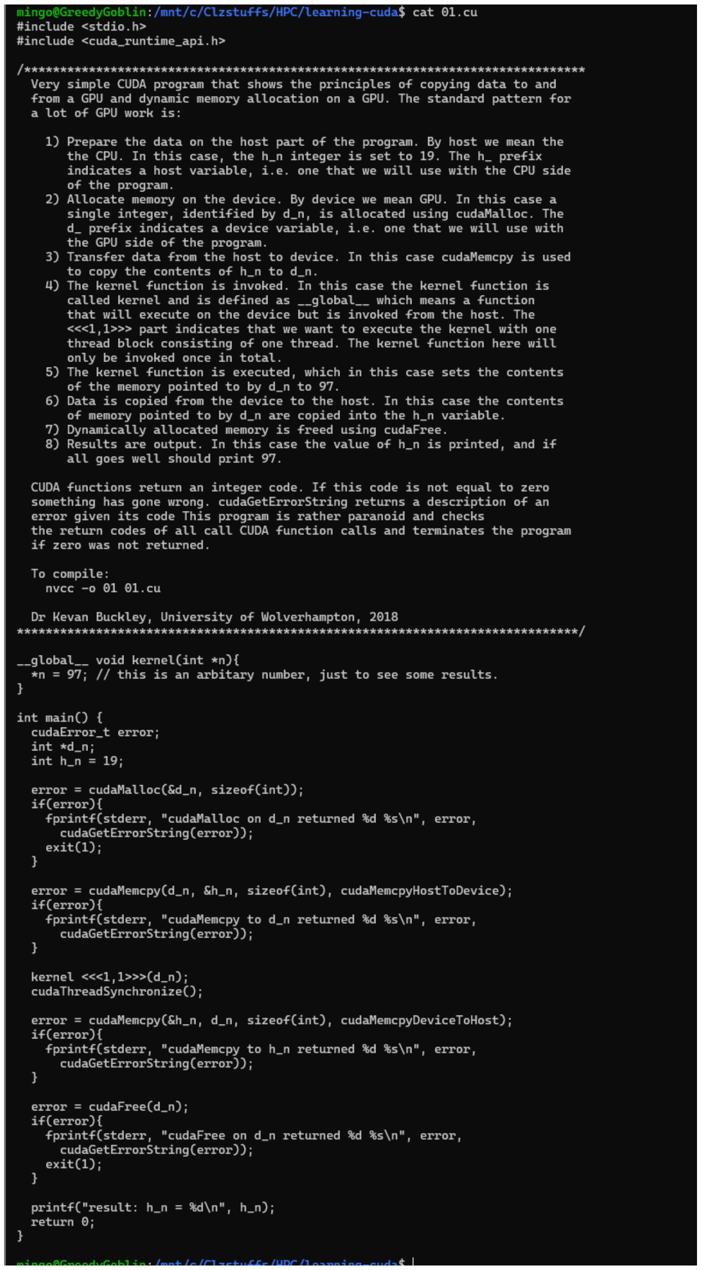
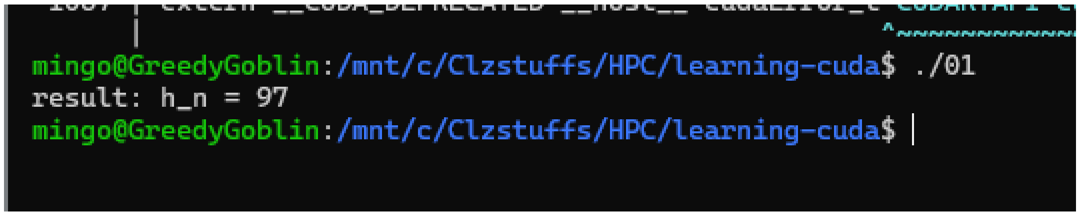
Week 8 zipn file (learning cuda)

1.

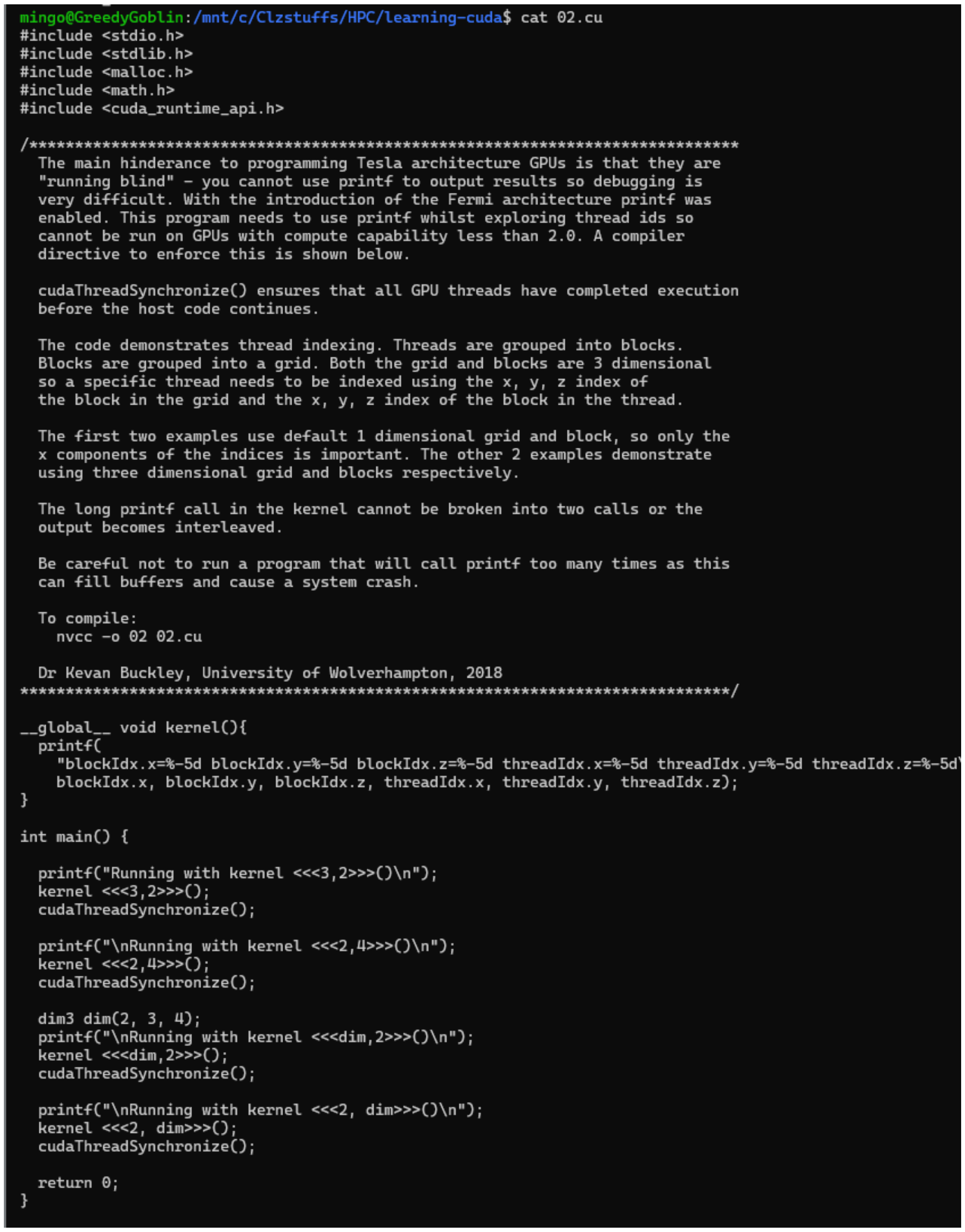


```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ cat 01.cu
#include <stdio.h>
#include <cuda_runtime_api.h>

/******************************************************************************
  Very simple CUDA program that shows the principles of copying data to and
  from a GPU and dynamic memory allocation on a GPU. The standard pattern for
  a lot of GPU work is:

    1) Prepare the data on the host part of the program. By host we mean the
       the CPU. In this case, the h_n integer is set to 19. The h_ prefix
       indicates a host variable, i.e. one that we will use with the CPU side
       of the program.
    2) Allocate memory on the device. By device we mean GPU. In this case a
       single integer, identified by d_n, is allocated using cudaMalloc. The
       d_ prefix indicates a device variable, i.e. one that we will use with
       the GPU side of the program.
    3) Transfer data from the host to device. In this case cudaMemcpy is used
       to copy the contents of h_n to d_n.
    4) The kernel function is invoked. In this case the kernel function is
       called kernel and is defined as __global__ which means a function
       that will execute on the device but is invoked from the host. The
       <<<1,1>>> part indicates that we want to execute the kernel with one
       thread block consisting of one thread. The kernel function here will
       only be invoked once in total.
    5) The kernel function is executed, which in this case sets the contents
       of the memory pointed to by d_n to 97.
    6) Data is copied from the device to the host. In this case the contents
       of memory pointed to by d_n are copied into the h_n variable.
    7) Dynamically allocated memory is freed using cudaFree.
    8) Results are output. In this case the value of h_n is printed, and if
       all goes well should print 97.

  CUDA functions return an integer code. If this code is not equal to zero
  something has gone wrong. cudaGetErrorString returns a description of an
  error given its code This program is rather paranoid and checks
  the return codes of all call CUDA function calls and terminates the program
  if zero was not returned.

  To compile:
    nvcc -o 01 01.cu

  Dr Kevan Buckley, University of Wolverhampton, 2018
******************************************************************************/

__global__ void kernel(int *n){
  *n = 97; // this is an arbitary number, just to see some results.
}

int main() {
  cudaError_t error;
  int *d_n;
  int h_n = 19;

  error = cudaMalloc(&d_n, sizeof(int));
  if(error){
    fprintf(stderr, "cudaMalloc on d_n returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMemcpy(d_n, &h_n, sizeof(int), cudaMemcpyHostToDevice);
  if(error){
    fprintf(stderr, "cudaMemcpy to d_n returned %d %s\n", error,
      cudaGetErrorString(error));
  }

  kernel <<<1,1>>>(d_n);
  cudaThreadSynchronize();

  error = cudaMemcpy(&h_n, d_n, sizeof(int), cudaMemcpyDeviceToHost);
  if(error){
    fprintf(stderr, "cudaMemcpy to h_n returned %d %s\n", error,
      cudaGetErrorString(error));
  }

  error = cudaFree(d_n);
  if(error){
    fprintf(stderr, "cudaFree on d_n returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  printf("result: h_n = %d\n", h_n);
  return 0;
}
```

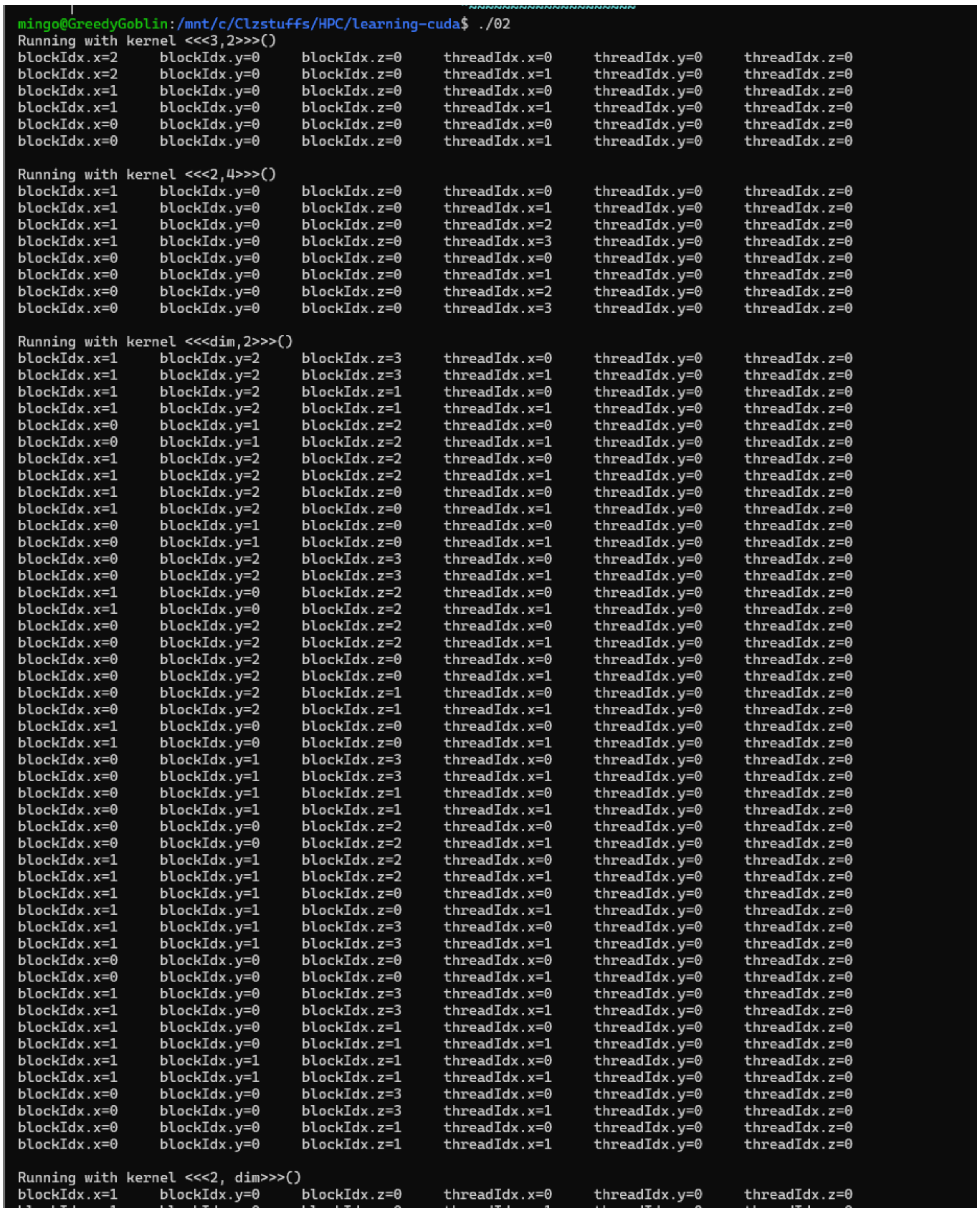


```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ ./01
result: h_n = 97
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$
```

This CUDA program demonstrates a basic workflow for using a GPU. First, you prepare data on the CPU (host) and allocate matching space on the GPU (device). You then copy the data from the CPU to the GPU and run a special function called a "kernel" to process it. Once the GPU finishes its task, the results are copied back to the CPU, and the GPU memory is cleaned up. Throughout the process, the program checks for errors at every step to ensure the hardware is communicating correctly, ultimately displaying the final result.

2. 

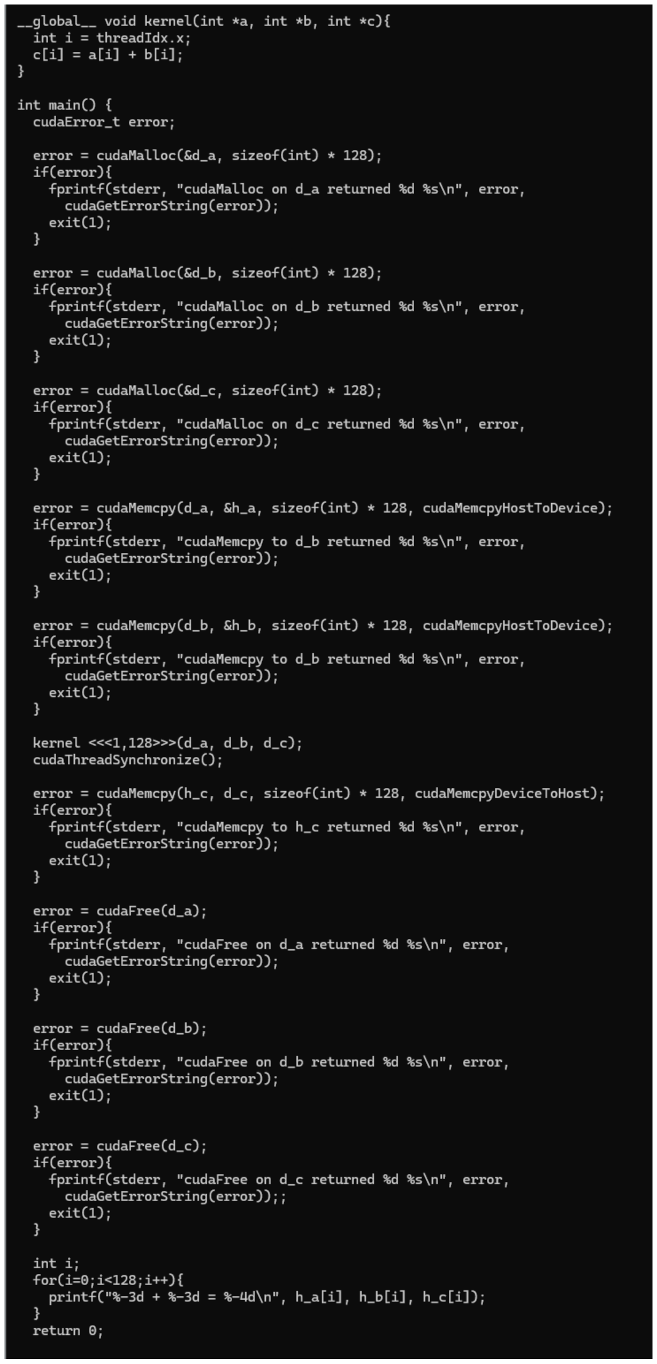
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ cat 02.cu
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include <cuda_runtime_api.h>

/******************************************************************************
  The main hinderance to programming Tesla architecture GPUs is that they are
  "running blind" - you cannot use printf to output results so debugging is
  very difficult. With the introduction of the Fermi architecture printf was
  enabled. This program needs to use printf whilst exploring thread ids so
  cannot be run on GPUs with compute capability less than 2.0. A compiler
  directive to enforce this is shown below.

  cudaThreadSynchronize() ensures that all GPU threads have completed execution
  before the host code continues.

  The code demonstrates thread indexing. Threads are grouped into blocks.
  Blocks are grouped into a grid. Both the grid and blocks are 3 dimensional
  so a specific thread needs to be indexed using the x, y, z index of
  the block in the grid and the x, y, z index of the block in the thread.

  The first two examples use default 1 dimensional grid and block, so only the
  x components of the indices is important. The other 2 examples demonstrate
  using three dimensional grid and blocks respectively.

  The long printf call in the kernel cannot be broken into two calls or the
  output becomes interleaved.

  Be careful not to run a program that will call printf too many times as this
  can fill buffers and cause a system crash.

  To compile:
    nvcc -o 02 02.cu

  Dr Kevan Buckley, University of Wolverhampton, 2018
*****************************************************************************/

__global__ void kernel(){
  printf(
    "blockIdx.x=%-5d blockIdx.y=%-5d blockIdx.z=%-5d threadIdx.x=%-5d threadIdx.y=%-5d threadIdx.z=%-5d
    blockIdx.x, blockIdx.y, blockIdx.z, threadIdx.x, threadIdx.y, threadIdx.z);
}

int main() {

  printf("Running with kernel <<<3,2>>>()\n");
  kernel <<<3,2>>>();
  cudaThreadSynchronize();

  printf("\nRunning with kernel <<<2,4>>>()\n");
  kernel <<<2,4>>>();
  cudaThreadSynchronize();

  dim3 dim(2, 3, 4);
  printf("\nRunning with kernel <<<dim,2>>>()\n");
  kernel <<<dim,2>>>();
  cudaThreadSynchronize();

  printf("\nRunning with kernel <<<2, dim>>>()\n");
  kernel <<<2, dim>>>();
  cudaThreadSynchronize();

  return 0;
}
```



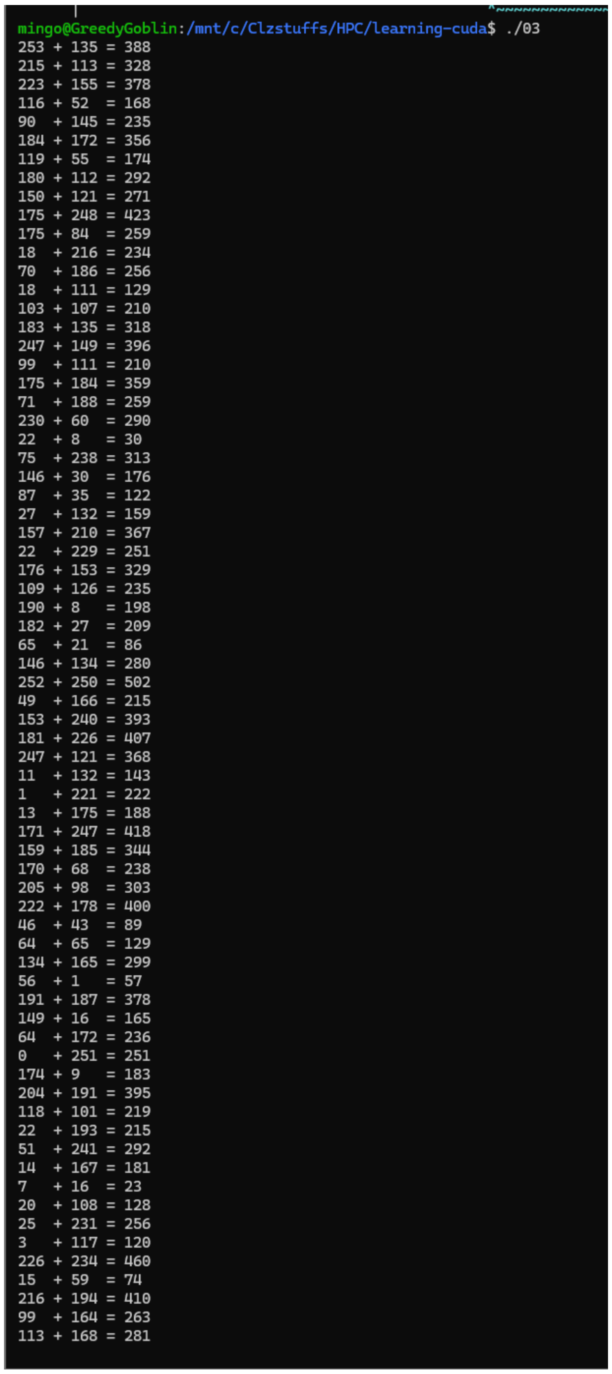
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ ./02
Running with kernel <<<3,2>>>()
blockIdx.x=2     blockIdx.y=0     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=2     blockIdx.y=0     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0

Running with kernel <<<2,4>>>()
blockIdx.x=1     blockIdx.y=0     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=0     threadIdx.x=2     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=0     threadIdx.x=3     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=0     threadIdx.x=2     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=0     threadIdx.x=3     threadIdx.y=0     threadIdx.z=0

Running with kernel <<<dim,2>>>()
blockIdx.x=1     blockIdx.y=2     blockIdx.z=3     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=2     blockIdx.z=3     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=2     blockIdx.z=1     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=2     blockIdx.z=1     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=1     blockIdx.z=2     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=1     blockIdx.z=2     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=2     blockIdx.z=2     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=2     blockIdx.z=2     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=2     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=2     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=1     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=1     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=2     blockIdx.z=3     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=2     blockIdx.z=3     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=2     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=2     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=2     blockIdx.z=2     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=2     blockIdx.z=2     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=2     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=2     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=2     blockIdx.z=1     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=2     blockIdx.z=1     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=1     blockIdx.z=3     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=1     blockIdx.z=3     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=1     blockIdx.z=1     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=1     blockIdx.z=1     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=2     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=2     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=1     blockIdx.z=2     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=1     blockIdx.z=2     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=1     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=1     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=1     blockIdx.z=3     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=1     blockIdx.z=3     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=0     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=3     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=3     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=1     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=0     blockIdx.z=1     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=1     blockIdx.z=1     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=1     blockIdx.y=1     blockIdx.z=1     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=3     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=3     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=1     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
blockIdx.x=0     blockIdx.y=0     blockIdx.z=1     threadIdx.x=1     threadIdx.y=0     threadIdx.z=0

Running with kernel <<<2, dim>>>()
blockIdx.x=1     blockIdx.y=0     blockIdx.z=0     threadIdx.x=0     threadIdx.y=0     threadIdx.z=0
```

Newer GPUs allow programmers to use the "printf" command to see what is happening inside their code, which makes debugging much easier than on older models. This program uses that feature to show how GPU threads are organized into 3D groups called blocks and grids. By using specific coordinates, the program can identify every individual thread and track its work. It also uses a synchronization command to make sure the CPU waits for the GPU to finish everything before moving forward. While this is a great way to learn how threads work, it is important not to print too much information at once, as it can overload the system and cause a crash.

3.



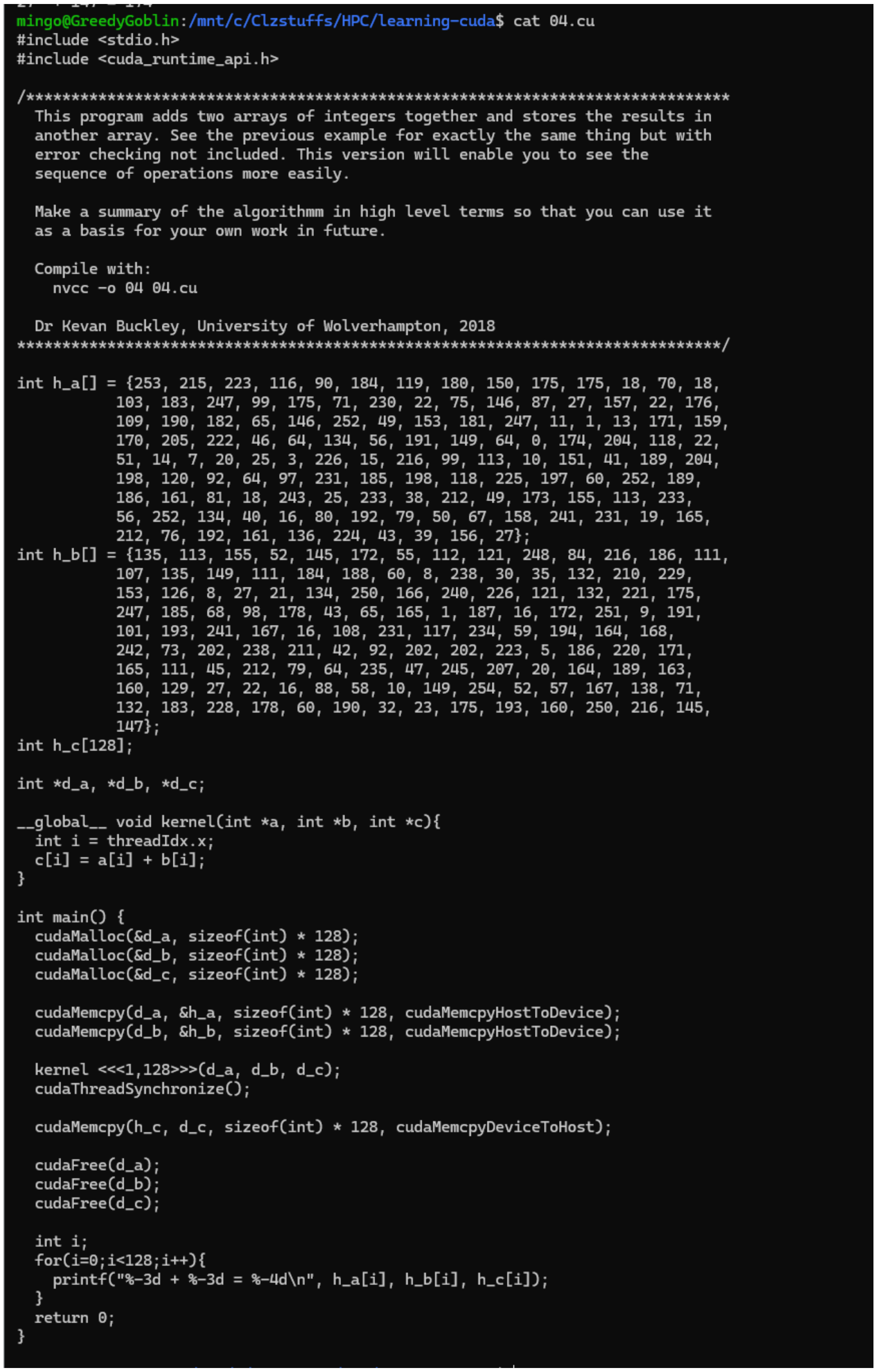
```
__global__ void kernel(int *a, int *b, int *c){
  int i = threadIdx.x;
  c[i] = a[i] + b[i];
}

int main() {
  cudaError_t error;

  error = cudaMalloc(&d_a, sizeof(int) * 128);
  if(error){
    fprintf(stderr, "cudaMalloc on d_a returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMalloc(&d_b, sizeof(int) * 128);
  if(error){
    fprintf(stderr, "cudaMalloc on d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMalloc(&d_c, sizeof(int) * 128);
  if(error){
    fprintf(stderr, "cudaMalloc on d_c returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMemcpy(d_a, &h_a, sizeof(int) * 128, cudaMemcpyHostToDevice);
  if(error){
    fprintf(stderr, "cudaMemcpy to d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMemcpy(d_b, &h_b, sizeof(int) * 128, cudaMemcpyHostToDevice);
  if(error){
    fprintf(stderr, "cudaMemcpy to d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  kernel <<<1,128>>>(d_a, d_b, d_c);
  cudaThreadSynchronize();

  error = cudaMemcpy(h_c, d_c, sizeof(int) * 128, cudaMemcpyDeviceToHost);
  if(error){
    fprintf(stderr, "cudaMemcpy to h_c returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaFree(d_a);
  if(error){
    fprintf(stderr, "cudaFree on d_a returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaFree(d_b);
  if(error){
    fprintf(stderr, "cudaFree on d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaFree(d_c);
  if(error){
    fprintf(stderr, "cudaFree on d_c returned %d %s\n", error,
      cudaGetErrorString(error));;
    exit(1);
  }

  int i;
  for(i=0;i<128;i++){
    printf("%-3d + %-3d = %-4d\n", h_a[i], h_b[i], h_c[i]);
  }
  return 0;
```



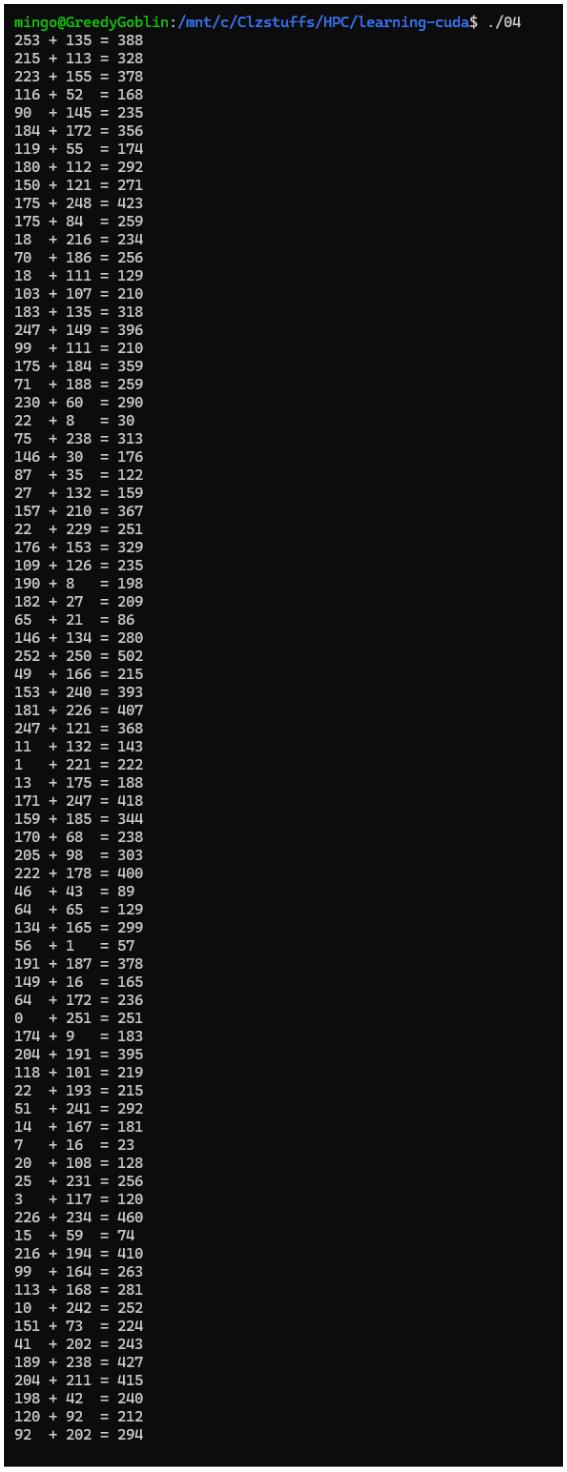
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ ./03
253 + 135 = 388
215 + 113 = 328
223 + 155 = 378
116 + 52  = 168
90  + 145 = 235
184 + 172 = 356
119 + 55  = 174
180 + 112 = 292
150 + 121 = 271
175 + 248 = 423
175 + 84  = 259
18  + 216 = 234
70  + 186 = 256
18  + 111 = 129
103 + 107 = 210
183 + 135 = 318
247 + 149 = 396
99  + 111 = 210
175 + 184 = 359
71  + 188 = 259
230 + 60  = 290
22  + 8   = 30
75  + 238 = 313
146 + 30  = 176
87  + 35  = 122
27  + 132 = 159
157 + 210 = 367
22  + 229 = 251
176 + 153 = 329
109 + 126 = 235
190 + 8   = 198
182 + 27  = 209
65  + 21  = 86
146 + 134 = 280
252 + 250 = 502
49  + 166 = 215
153 + 240 = 393
181 + 226 = 407
247 + 121 = 368
11  + 132 = 143
1   + 221 = 222
13  + 175 = 188
171 + 247 = 418
159 + 185 = 344
170 + 68  = 238
205 + 98  = 303
222 + 178 = 400
46  + 43  = 89
64  + 65  = 129
134 + 165 = 299
56  + 1   = 57
191 + 187 = 378
149 + 16  = 165
64  + 172 = 236
0   + 251 = 251
174 + 9   = 183
204 + 191 = 395
118 + 101 = 219
22  + 193 = 215
51  + 241 = 292
14  + 167 = 181
7   + 16  = 23
20  + 108 = 128
25  + 231 = 256
3   + 117 = 120
226 + 234 = 460
15  + 59  = 74
216 + 194 = 410
99  + 164 = 263
113 + 168 = 281
```

This program demonstrates how to add two lists of numbers together using the GPU. Because the lists are small, the work is handled by a single group of threads. The process involves setting up memory on the GPU, copying the input data over, and then copying only the final results back to the computer once the calculation is done. A simplified version of this code is also available without error-checking to make the core steps easier to see.

4.



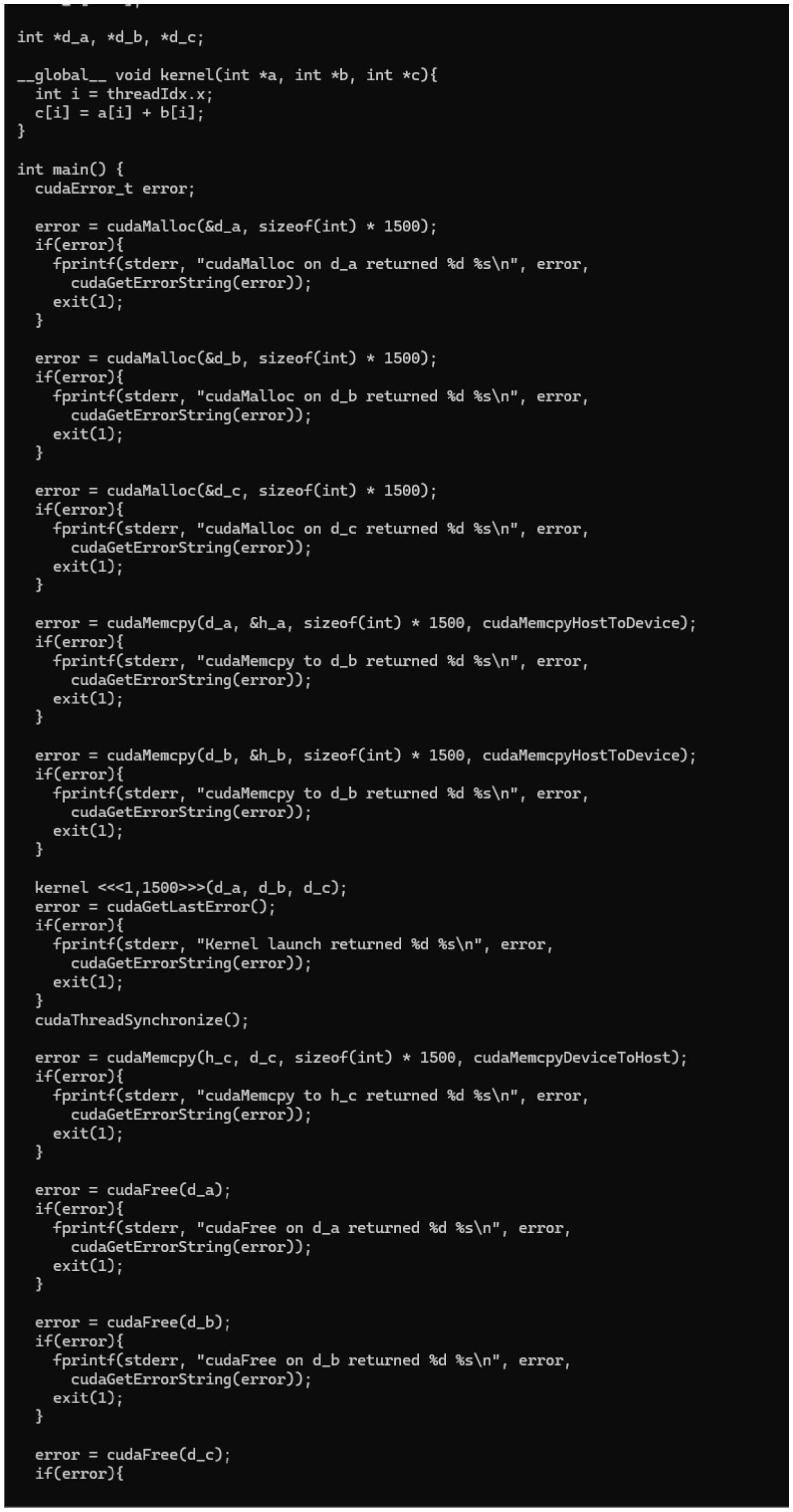
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ cat 04.cu
#include <stdio.h>
#include <cuda_runtime_api.h>

/******************************************************************************
  This program adds two arrays of integers together and stores the results in
  another array. See the previous example for exactly the same thing but with
  error checking not included. This version will enable you to see the
  sequence of operations more easily.

  Make a summary of the algorithmm in high level terms so that you can use it
  as a basis for your own work in future.

  Compile with:
    nvcc -o 04 04.cu

  Dr Kevan Buckley, University of Wolverhampton, 2018
******************************************************************************/

int h_a[] = {253, 215, 223, 116, 90, 184, 119, 180, 150, 175, 175, 18, 70, 18,
            103, 183, 247, 99, 175, 71, 230, 22, 75, 146, 87, 27, 157, 22, 176,
            109, 190, 182, 65, 146, 252, 49, 153, 181, 247, 11, 1, 13, 171, 159,
            170, 205, 222, 46, 64, 134, 56, 191, 149, 64, 0, 174, 204, 118, 22,
            51, 14, 7, 20, 25, 3, 226, 15, 216, 99, 113, 10, 151, 41, 189, 204,
            198, 120, 92, 64, 97, 231, 185, 198, 118, 225, 197, 60, 252, 189,
            186, 161, 81, 18, 243, 25, 233, 38, 212, 49, 173, 155, 113, 233,
            56, 252, 134, 40, 16, 80, 192, 79, 50, 67, 158, 241, 231, 19, 165,
            212, 76, 192, 161, 136, 224, 43, 39, 156, 27};
int h_b[] = {135, 113, 155, 52, 145, 172, 55, 112, 121, 248, 84, 216, 186, 111,
            107, 135, 149, 111, 184, 188, 60, 8, 238, 30, 35, 132, 210, 229,
            153, 126, 8, 27, 21, 134, 250, 166, 240, 226, 121, 132, 221, 175,
            247, 185, 68, 98, 178, 43, 65, 165, 1, 187, 16, 172, 251, 9, 191,
            101, 193, 241, 167, 16, 108, 231, 117, 234, 59, 194, 164, 168,
            242, 73, 202, 238, 211, 42, 92, 202, 202, 223, 5, 186, 220, 171,
            165, 111, 45, 212, 79, 64, 235, 47, 245, 207, 20, 164, 189, 163,
            160, 129, 27, 22, 16, 88, 58, 10, 149, 254, 52, 57, 167, 138, 71,
            132, 183, 228, 178, 60, 190, 32, 23, 175, 193, 160, 250, 216, 145,
            147};
int h_c[128];

int *d_a, *d_b, *d_c;

__global__ void kernel(int *a, int *b, int *c){
  int i = threadIdx.x;
  c[i] = a[i] + b[i];
}

int main() {
  cudaMalloc(&d_a, sizeof(int) * 128);
  cudaMalloc(&d_b, sizeof(int) * 128);
  cudaMalloc(&d_c, sizeof(int) * 128);

  cudaMemcpy(d_a, &h_a, sizeof(int) * 128, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, &h_b, sizeof(int) * 128, cudaMemcpyHostToDevice);

  kernel <<<1,128>>>(d_a, d_b, d_c);
  cudaThreadSynchronize();

  cudaMemcpy(h_c, d_c, sizeof(int) * 128, cudaMemcpyDeviceToHost);

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  int i;
  for(i=0;i<128;i++){
    printf("%-3d + %-3d = %-4d\n", h_a[i], h_b[i], h_c[i]);
  }
  return 0;
}
```



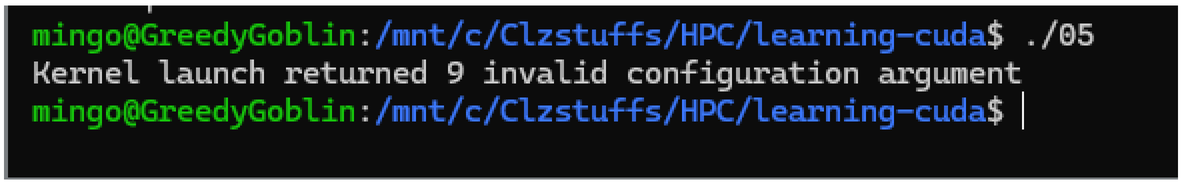
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ ./04
253 + 135 = 388
215 + 113 = 328
223 + 155 = 378
116 + 52  = 168
90  + 145 = 235
184 + 172 = 356
119 + 55  = 174
180 + 112 = 292
150 + 121 = 271
175 + 248 = 423
175 + 84  = 259
18  + 216 = 234
70  + 186 = 256
18  + 111 = 129
103 + 107 = 210
183 + 135 = 318
247 + 149 = 396
99  + 111 = 210
175 + 184 = 359
71  + 188 = 259
230 + 60  = 290
22  + 8   = 30
75  + 238 = 313
146 + 30  = 176
87  + 35  = 122
27  + 132 = 159
157 + 210 = 367
22  + 229 = 251
176 + 153 = 329
109 + 126 = 235
190 + 8   = 198
182 + 27  = 209
65  + 21  = 86
146 + 134 = 280
252 + 250 = 502
49  + 166 = 215
153 + 240 = 393
181 + 226 = 407
247 + 121 = 368
11  + 132 = 143
1   + 221 = 222
13  + 175 = 188
171 + 247 = 418
159 + 185 = 344
170 + 68  = 238
205 + 98  = 303
222 + 178 = 400
46  + 43  = 89
64  + 65  = 129
134 + 165 = 299
56  + 1   = 57
191 + 187 = 378
149 + 16  = 165
64  + 172 = 236
0   + 251 = 251
174 + 9   = 183
204 + 191 = 395
118 + 101 = 219
22  + 193 = 215
51  + 241 = 292
14  + 167 = 181
7   + 16  = 23
20  + 108 = 128
25  + 231 = 256
3   + 117 = 120
226 + 234 = 460
15  + 59  = 74
216 + 194 = 410
99  + 164 = 263
113 + 168 = 281
10  + 242 = 252
151 + 73  = 224
41  + 202 = 243
189 + 238 = 427
204 + 211 = 415
198 + 42  = 240
120 + 92  = 212
92  + 202 = 294
```

This program adds two lists of numbers together and saves the result in a third list. It is identical to the previous example but removes the error-checking code to make the steps easier to follow. By simplifying the code, you can more clearly see the exact sequence of operations the GPU uses to complete the task.

5.



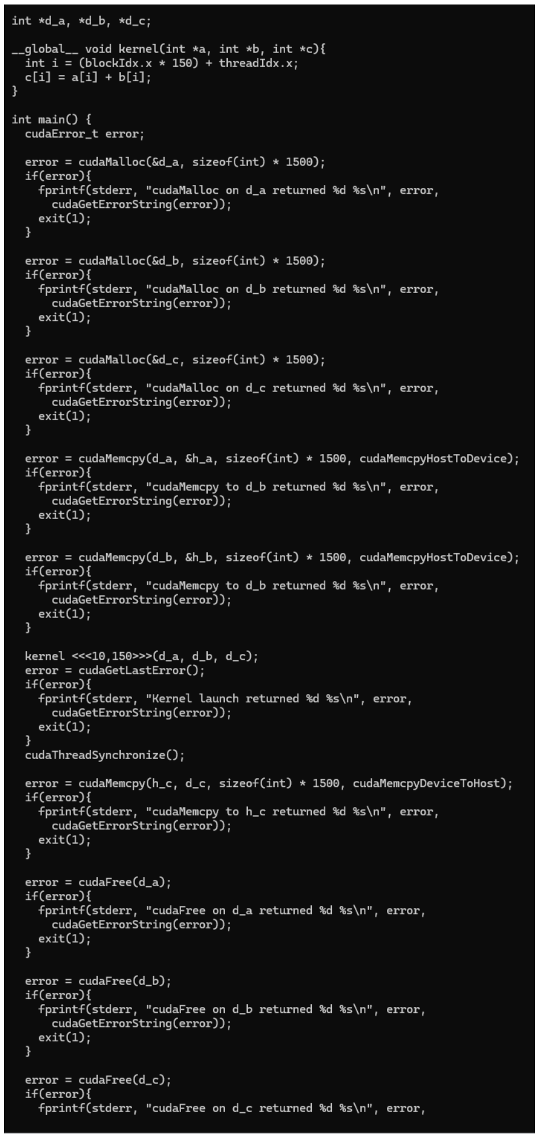
```
int *d_a, *d_b, *d_c;

__global__ void kernel(int *a, int *b, int *c){
  int i = threadIdx.x;
  c[i] = a[i] + b[i];
}

int main() {
  cudaError_t error;

  error = cudaMalloc(&d_a, sizeof(int) * 1500);
  if(error){
    fprintf(stderr, "cudaMalloc on d_a returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMalloc(&d_b, sizeof(int) * 1500);
  if(error){
    fprintf(stderr, "cudaMalloc on d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMalloc(&d_c, sizeof(int) * 1500);
  if(error){
    fprintf(stderr, "cudaMalloc on d_c returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMemcpy(d_a, &h_a, sizeof(int) * 1500, cudaMemcpyHostToDevice);
  if(error){
    fprintf(stderr, "cudaMemcpy to d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMemcpy(d_b, &h_b, sizeof(int) * 1500, cudaMemcpyHostToDevice);
  if(error){
    fprintf(stderr, "cudaMemcpy to d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  kernel <<<1,1500>>>(d_a, d_b, d_c);
  error = cudaGetLastError();
  if(error){
    fprintf(stderr, "Kernel launch returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }
  cudaThreadSynchronize();

  error = cudaMemcpy(h_c, d_c, sizeof(int) * 1500, cudaMemcpyDeviceToHost);
  if(error){
    fprintf(stderr, "cudaMemcpy to h_c returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaFree(d_a);
  if(error){
    fprintf(stderr, "cudaFree on d_a returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaFree(d_b);
  if(error){
    fprintf(stderr, "cudaFree on d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaFree(d_c);
  if(error){
```



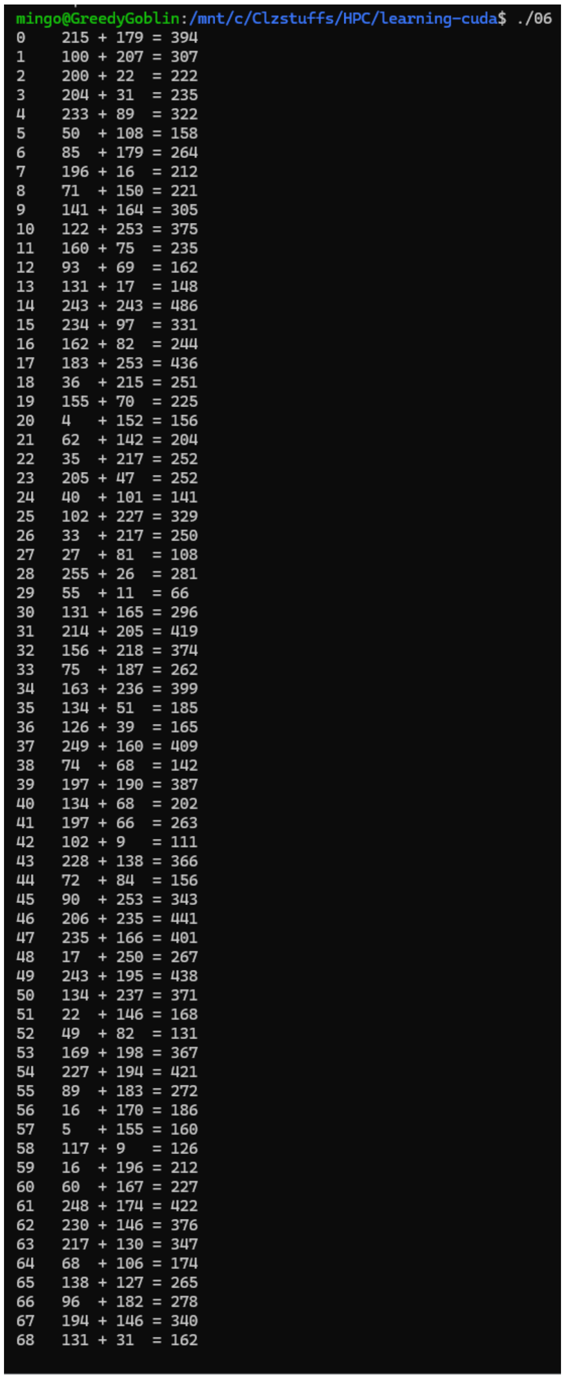
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ ./05
Kernel launch returned 9 invalid configuration argument
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$
```

This program attempts to add two large lists of numbers, but it will likely fail because the data is too big for a single group of threads. Since most GPUs have a limit of 512 threads per block, this code is designed to show you what a "kernel launch failure" looks like when you exceed that limit. Additionally, the program is purposely inefficient because it doesn't clean up its used memory, resulting in a memory leak.

6.



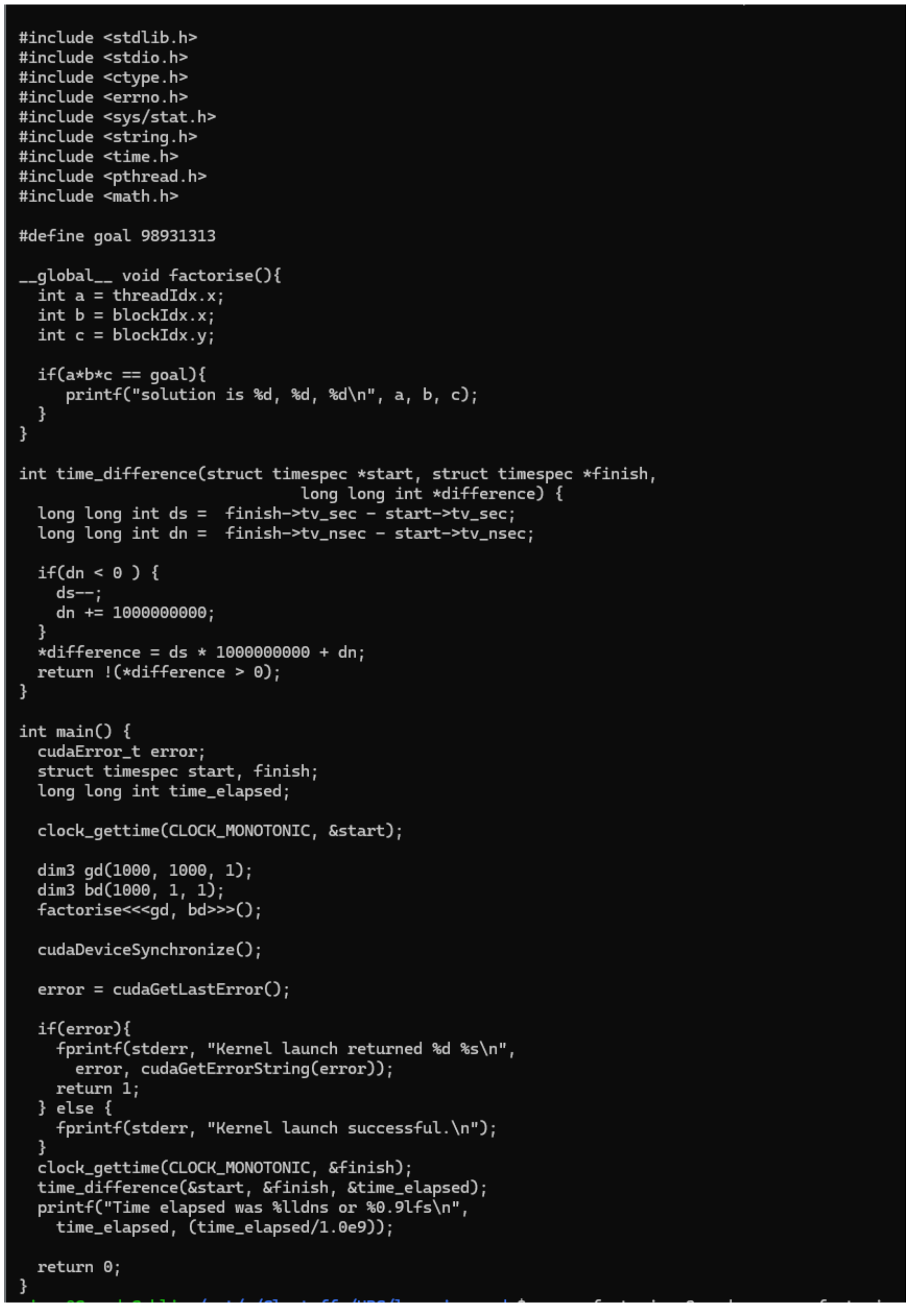
```
int *d_a, *d_b, *d_c;

__global__ void kernel(int *a, int *b, int *c){
  int i = (blockIdx.x * 150) + threadIdx.x;
  c[i] = a[i] + b[i];
}

int main() {
  cudaError_t error;

  error = cudaMalloc(&d_a, sizeof(int) * 1500);
  if(error){
    fprintf(stderr, "cudaMalloc on d_a returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMalloc(&d_b, sizeof(int) * 1500);
  if(error){
    fprintf(stderr, "cudaMalloc on d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMalloc(&d_c, sizeof(int) * 1500);
  if(error){
    fprintf(stderr, "cudaMalloc on d_c returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMemcpy(d_a, &h_a, sizeof(int) * 1500, cudaMemcpyHostToDevice);
  if(error){
    fprintf(stderr, "cudaMemcpy to d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaMemcpy(d_b, &h_b, sizeof(int) * 1500, cudaMemcpyHostToDevice);
  if(error){
    fprintf(stderr, "cudaMemcpy to d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  kernel <<<10,150>>>(d_a, d_b, d_c);
  error = cudaGetLastError();
  if(error){
    fprintf(stderr, "Kernel launch returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }
  cudaThreadSynchronize();

  error = cudaMemcpy(h_c, d_c, sizeof(int) * 1500, cudaMemcpyDeviceToHost);
  if(error){
    fprintf(stderr, "cudaMemcpy to h_c returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaFree(d_a);
  if(error){
    fprintf(stderr, "cudaFree on d_a returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaFree(d_b);
  if(error){
    fprintf(stderr, "cudaFree on d_b returned %d %s\n", error,
      cudaGetErrorString(error));
    exit(1);
  }

  error = cudaFree(d_c);
  if(error){
    fprintf(stderr, "cudaFree on d_c returned %d %s\n", error,
```



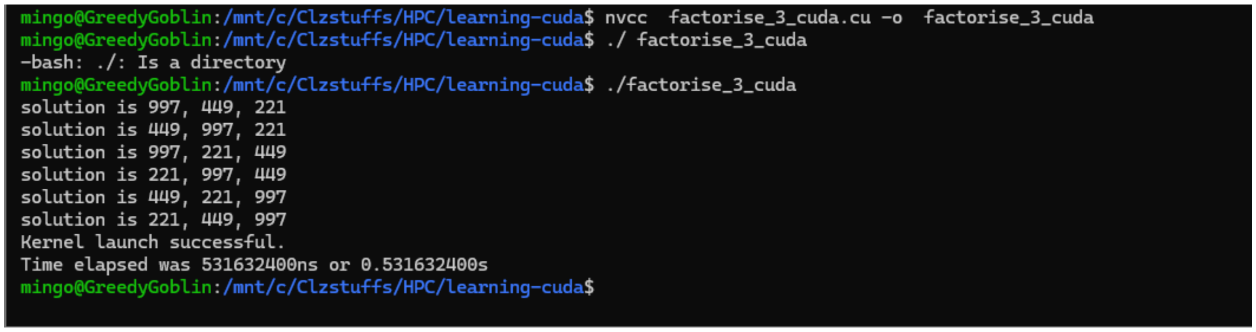
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ ./06
0    215 + 179 = 394
1    100 + 207 = 307
2    200 + 22  = 222
3    204 + 31  = 235
4    233 + 89  = 322
5    50  + 108 = 158
6    85  + 179 = 264
7    196 + 16  = 212
8    71  + 150 = 221
9    141 + 164 = 305
10   122 + 253 = 375
11   160 + 75  = 235
12   93  + 69  = 162
13   131 + 17  = 148
14   243 + 243 = 486
15   234 + 97  = 331
16   162 + 82  = 244
17   183 + 253 = 436
18   36  + 215 = 251
19   155 + 70  = 225
20   4   + 152 = 156
21   62  + 142 = 204
22   35  + 217 = 252
23   205 + 47  = 252
24   40  + 101 = 141
25   102 + 227 = 329
26   33  + 217 = 250
27   27  + 81  = 108
28   255 + 26  = 281
29   55  + 11  = 66
30   131 + 165 = 296
31   214 + 205 = 419
32   156 + 218 = 374
33   75  + 187 = 262
34   163 + 236 = 399
35   134 + 51  = 185
36   126 + 39  = 165
37   249 + 160 = 409
38   74  + 68  = 142
39   197 + 190 = 387
40   134 + 68  = 202
41   197 + 66  = 263
42   102 + 9   = 111
43   228 + 138 = 366
44   72  + 84  = 156
45   90  + 253 = 343
46   206 + 235 = 441
47   235 + 166 = 401
48   17  + 250 = 267
49   243 + 195 = 438
50   134 + 237 = 371
51   22  + 146 = 168
52   49  + 82  = 131
53   169 + 198 = 367
54   227 + 194 = 421
55   89  + 183 = 272
56   16  + 170 = 186
57   5   + 155 = 160
58   117 + 9   = 126
59   16  + 196 = 212
60   60  + 167 = 227
61   248 + 174 = 422
62   230 + 146 = 376
63   217 + 130 = 347
64   68  + 106 = 174
65   138 + 127 = 265
66   96  + 182 = 278
67   194 + 146 = 340
68   131 + 31  = 162
```

This program adds two large lists of numbers together while carefully staying within the GPU's thread limits. Because the data is so big, the program calculates a unique ID for every individual thread to ensure each one knows exactly which piece of information to process. This approach allows the GPU to handle massive amounts of data correctly by splitting the work across many organized groups of threads.

factorise_3_cuda.



```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <sys/stat.h>
#include <string.h>
#include <time.h>
#include <pthread.h>
#include <math.h>

#define goal 98931313

__global__ void factorise(){
  int a = threadIdx.x;
  int b = blockIdx.x;
  int c = blockIdx.y;

  if(a*b*c == goal){
     printf("solution is %d, %d, %d\n", a, b, c);
  }
}

int time_difference(struct timespec *start, struct timespec *finish,
                              long long int *difference) {
  long long int ds =  finish->tv_sec - start->tv_sec;
  long long int dn =  finish->tv_nsec - start->tv_nsec;

  if(dn < 0 ) {
    ds--;
    dn += 1000000000;
  }
  *difference = ds * 1000000000 + dn;
  return !(*difference > 0);
}

int main() {
  cudaError_t error;
  struct timespec start, finish;
  long long int time_elapsed;

  clock_gettime(CLOCK_MONOTONIC, &start);

  dim3 gd(1000, 1000, 1);
  dim3 bd(1000, 1, 1);
  factorise<<<gd, bd>>>();

  cudaDeviceSynchronize();

  error = cudaGetLastError();

  if(error){
    fprintf(stderr, "Kernel launch returned %d %s\n",
      error, cudaGetErrorString(error));
    return 1;
  } else {
    fprintf(stderr, "Kernel launch successful.\n");
  }
  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &time_elapsed);
  printf("Time elapsed was %lldns or %0.9lfs\n",
    time_elapsed, (time_elapsed/1.0e9));

  return 0;
}
```



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ nvcc  factorise_3_cuda.cu -o  factorise_3_cuda
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ ./ factorise_3_cuda
-bash: ./: Is a directory
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$ ./factorise_3_cuda
solution is 997, 449, 221
solution is 449, 997, 221
solution is 997, 221, 449
solution is 221, 997, 449
solution is 449, 221, 997
solution is 221, 449, 997
Kernel launch successful.
Time elapsed was 531632400ns or 0.531632400s
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/learning-cuda$
```

This program performs the same task of breaking down numbers into factors as the previous versions we studied with CPU threads. The main difference is that this version is designed to run on a GPU using CUDA, allowing the graphics card to handle the mathematical calculations instead of the main processor.