Cheat Sheet

Programming excersise

Matrix multiplication

```
#define Size 256

void main() {
  float N[Size][Size], M[Size][Size], P[Size][Size];
  int i, j, k;
  GetMatrix(N, file1); GetMatrix(M, file2); /* Read N and M */
  for(i=0;i<Size;i++) {
    for(j=0;j<Size;j++) {
      P[i][j]=0;
      for(k=0;k<Size;k++)
        P[i][j]+=M[i][k]*N[k][j];
     }
   }
   PutMatrix(P, file3); /* Skriv ut P */
}
```

Modify the sequential algorithm to be performed in parallel using CUDA. Also use shared memory.

```
void perform(float** N, float** M, float** P) {
  float *dev_N = 0;
  float *dev_M = 0;
  float *dev_P = 0;
  cudaMalloc(&dev_N, sizeof(float) * Size * Size);
  cudaMalloc(&dev_M, sizeof(float) * Size * Size);
  cudaMalloc(&dev_P, sizeof(float) * Size * Size);
  for (int row = 0; row < Size; row++) {
    cudaMemcpy(dev_N + row * Size, N[row], sizeof(float) * Size,
cudaMemcpyHostToDevice)
    cudaMemcpy(dev_M + row * Size, M[row], sizeof(float) * Size,
cudaMemcpyHostToDevice)
  }

  int threads = Size * Size;
  int blocks = ceil(threads / (double)1024);
  int threadsPerBlock = min(ceil(threads / blocks), 1024);
  kernel<<<blocks, threadsPerBlock>>>(dev_N, dev_M, dev_P);

  cudaDeviceSynchronize();
```

```
  for (int row = 0; row < Size; row++)
    cudaMemcpy(P[row], dev_P + row * Size, sizeof(float) * Size,
cudaMemcpyDeviceToHost);

  cudaFree(dev_N);
  cudaFree(dev_M);
  cudaFree(dev_P);
}
```

```
__global__ void kernel(float *dev_N, float* dev_M, float *dev_P) {
  __shared__ float shared[1024] = {0};
  long threadId = blockIdx.x * blockDim.x + threadIdx.x;
  int i = threadId / Size;
  int j = threadId % Size;
  if (i >= Size || j >= Size)
    return;
  shared[threadIdx.x] = dev_M[threadId];
  __syncthreads();
  for(int k = 0; k < Size; k++)
    dev_P[i * Size + j] += shared[(i * size) % Size + k] * dev_N[k * size + j];
}
```

Cumulative sum

```
#define Size 100

void main() {
  int in_vect[Size]; out_vect[Size]
  int i;
  GetVect(in_vect, file1); /* Read the in-vector */
  out_vect[0]= in_vect[0]
  for (i=1;i<Size;i++)
   out_vect[i] = out_vect[i-1] + in_vect[i];
  PutVect(out_vect, file2); /* Write the out-vector */
}
```

Rewrite (modify) this program in a host and a device part so that the program benefits from a graphical card that supports CUDA. You can assume that all data structures fit in the graphical card's memory. Please explain all the parameters that you use. You should minimize thread divergence. Also, explain what thread divergence is and how you have tried to minimize it.

```
void perform(int *in_vect, int *out_vect) {
  int *dev_vect = 0;

  cudaMalloc(&dev_vect, sizeof(int) * Size);
  cudaMemcpy(dev_vect, in_vect, sizeof(int) * Size, cudaMemcpyHostToDevice);
```

```
  // We're using __syncthreads() heavily, so supporting multiple blocks is non-
trivial.
  // Mitigate by wrapping the kernel calls in a for loop instead
  forwardKernel<<<1, Size>>>(dev_vect);
  backwardKernel<<<1, Size>>>(dev_vect);

  cudaDeviceSynchronize();

  cudaMemcpy(out_vect, dev_vect, sizeof(int) * Size, cudaMemcpyDeviceToHost);

  cudaFree(dev_in_vect);
  cudaFree(dev_out_vect);
}
```

```
__global__ void forwardKernel(int *dev_vect) {
  for (int stride = 1; stride < Size; stride *= 2) {
    int index = (threadIdx.x + 1) * stride * 2 - 1;
    if (index < Size)
      dev_vect[index] += dev_vect[index - stride];
    __syncthreads();
  }
}
```

```
__global__ void backwardKernel(int *dev_vect) {
  for (int stride = Size / 2; stride > 0; stride /= 2) {
    int index = (threadIdx.x + 1) * stride * 2 - 1;
    if (index < Size)
      dev_vect[index + stride] += dev_vect[index];
    __syncthreads();
  }
}
```

To prehibit thread divergence we have ensured that there are no if-else-cases. The only potential divergence in this code is the if statement. In this case, some threads will continue to calculate values for `dev_vect`, the others will just wait. This is a non-divergent behaviour. A divergent behaviour causing performance issues would be if the threads not entering the true if branch would calculate other values. TLDR; the threads not "accepted" in the if case will sync.

See also the optimization techniques.

Memory management

CUDA

What types of memory are supported in CUDA?

What is the scope for the different memory types in terms of thread/block/grid?

What is the lifetime of the differerent types of memory?

How can a programmer control the type of memory that is used for different variables?

- global (prefix: `__global__`)
  - Use case: variables shared between all threads in the kernel
  - Scope: grid
  - Lifetime: kernel
  - Required cycles: ~500
- shared (prefix: `__shared__`)
  - Use case: variables shared between threads of the same block in the kernel
  - Scope: block
  - Lifetime: block
  - Required cycles: ~5
- local (prefix: `__local__`)
  - Use case: automatically used for thread-local arrays that cannot fit in registers. Part of the global memory, addressable by the thread.
  - Scope: thread
  - Lifetime: thread
  - Required cycles: ~500
- register (prefix: `__register__`)
  - Use case: variables in kernel
  - Scope: thread
  - Lifetime: thread
  - Required cycles: ~1
- constant (prefix: `__constant__`)
  - Use case: constant values for entire kernel
  - Scope: grid
  - Lifetime: kernel
  - Required cycles: ~5

OpenCL

What types of memory are supported in OpenCL?

- global
- constant
- local (shared in CUDA)
- private (register in CUDA)

CUDA streams

CUDA streams enables a developer to execute functions concurrently on multiple processors. For example, in project 3, streams could be used to process each channel in an image in parallel. One can also use streams to start computation on the GPU and then continue doing other work on the CPU - enabling a pipeline.

Optimization techniques

Explain and give a short example of the following optimization techniques.

- Converting from scatter to gather
    - Scatter: each thread reads one value and writes to multiple outputs (slow due to synchronization)
    - Gather: each thread reads from multiple values and write to a single data unit (fast, no synchronization)
- Privatization
    - Histograms - sum values into a histogram in parallel and conclude them sequentially
    - Example: Counting the occurances of the letters in the alphabet in The Bible
- Thread coarsening
    - Let each thread perform more work. If you have a million of threads, reduce to a hundred thousand and let the threads perform 10x the work. The idea is to lower overhead
- Data layout transformation (to benefit from memory bursts)
    - Convert from array of structures to structures of arrays
    - Example: Project 3 is an example where the array of structure (array of RGB values) was converted to three arrays of values (one for R, G and B each)
- Software pipelining (out-of-order execution)
    - Enable the one to process data in parallel to other work being performed, for example by using streams
    - The idea is to process data while other data is fetched or being worked on sequentially (think doing laundry)
- Binning
    - Build data structures that map output locations to the relevant input data. Achieves gather (see converting from scatter to gather).
    - Example: bin ages into categories such as 0-10, 10-20, 20-30
    - Example: split a simulated volume into multiple parts
- Regularization
    - Better thread utilization. Adjacent threads may access non-adjacent memory locations
    - Reorganizing input data to reduce load imbalance. Go from irregular parallelism to regular parallelism
    - Example: Restructure data to ensure that each thread performs the same amount of work
- Padding
    - A type of regularization. Helps eliminate divirgence.
    - For access, shared memory is divided into equally-sized memory modules (banks) of 32 addresses. If two requested addresses are in the same bank, the request will be

serialized (a bank conflict occurs).
    - Example: declared as `__shared__ float x[32][32]` and accessed via `x[threadId.x][0]`. Will cause a 32-way bank conflict because all threads are accessing different values in the same bank. Instead we pad by declaring the memory as `float x[32][33]`, each bank location will now be offset by one.
- Transposition
    - Used to improve memory bursts by putting values used by a thread near eachother
    - Example: Transpose a matrix so that consecutive columns in a row are easily accessible
- Tiling (using shared memory)
    - By identifying a block or tile of global memory content that are access by multiple threads, one can load the block or tile into shared memory. Multiple threads can then use the shared memory for quicker access. Bottom line: reduce lookups in global memory.
    - Example: Matrix multiplication, copy a row to shared memory. See the code for the matrix multiplication programming exercise
- Minimize thread divergence
    - If-cases can cause thread divergence when some threads in a single warp evalute to "true" and others to "false". This means that the threads will want to proceed to different instructions. As all threads in a warp will execute the same instruction, divergent threads will slow down processing as it will require execution of instructions in sequence.
    - Example: Don't use 16 threads to do one operation and 16 to do another, for example (if-else etc.). All threads in a warp should use the same case or simply sync. That is, ensure each warp follows the same code branch or syncs.

Hardware

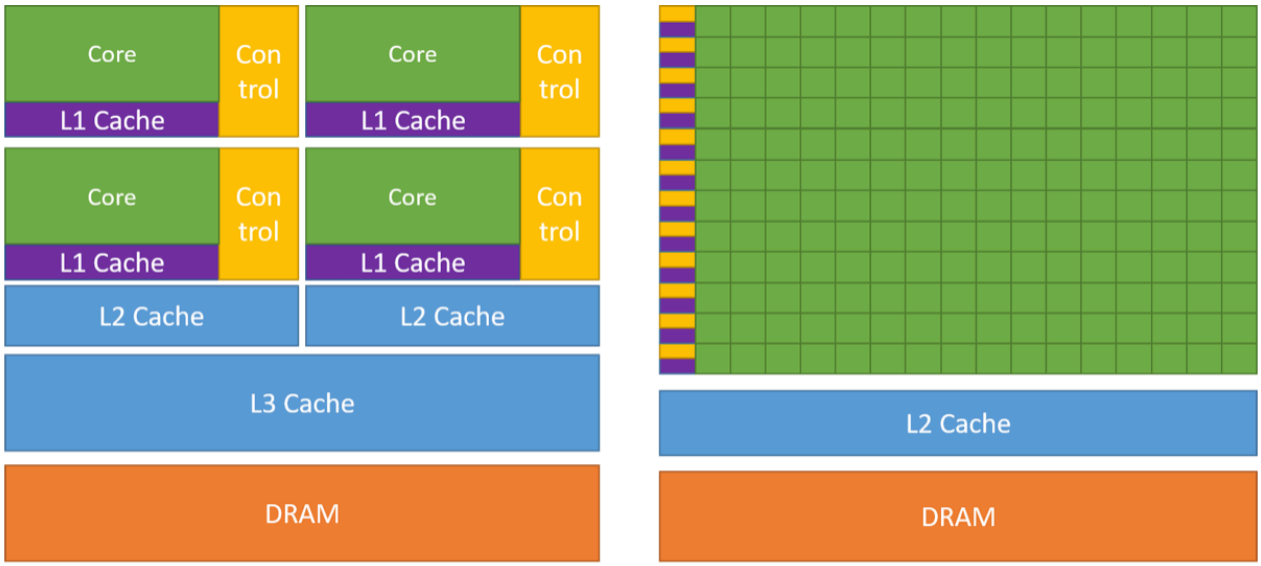
Architecture

GPUs and CPUs are designed in different ways. Explain the main differences and why they are designed differently.

GPUs are designed to perform highly parallel processing using techniques such as SIMD. CPUs are typically used for sequential processing.

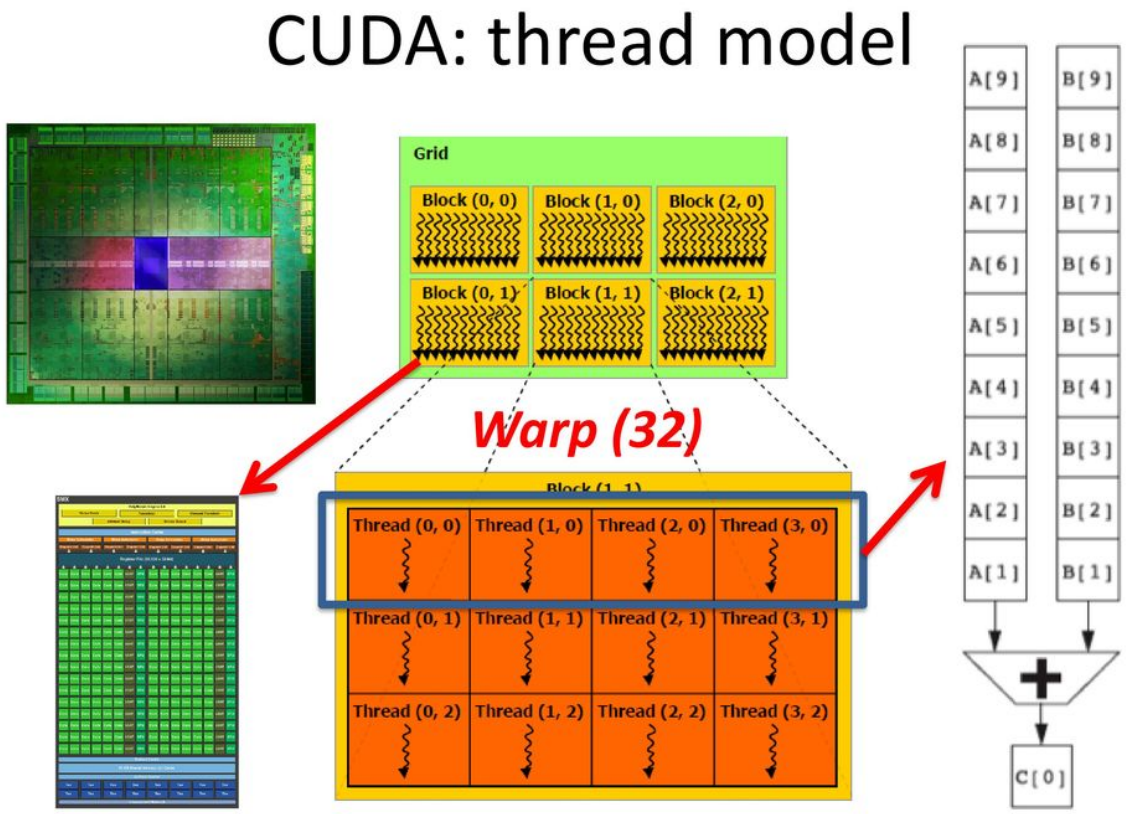
A CPU typically has slow memory access, but larger memory sizes. A GPU has much faster memory access, but usually a lower capacity. Both have different tiers of memory such as cache and global memory, but the use of which tier is much more controlled in a GPU program than a typical CPU program.

GPU cores are much less complex than CPU cores. They are designed to only perform a single task, comparable to the CPU's ALU. The GPU, however has thousands of cores and hundreds of thousands of threads whilst the CPU typically has around six cores.



Warps

A warp is a collection of 32 threads within a thread block that all execute the same instruction. These warps are handled by a warp scheduler. The scheduler can use multiple modes such as round robin, fair, least recently fetched, thread-block-based Critical Aware Warp-Scheduling (CAWS).



Syncing

Explain how the function `__synchthreads()` works and when we need a function like this.

The `__syncthreads()` is a block-level synchronization barrier. That is, all threads in a block wait for eachother to finish up until that point before continuing doing work.

The function can be used, for example when all threads require a certain value to be calculated before continuing - basically to resolve dependencies.

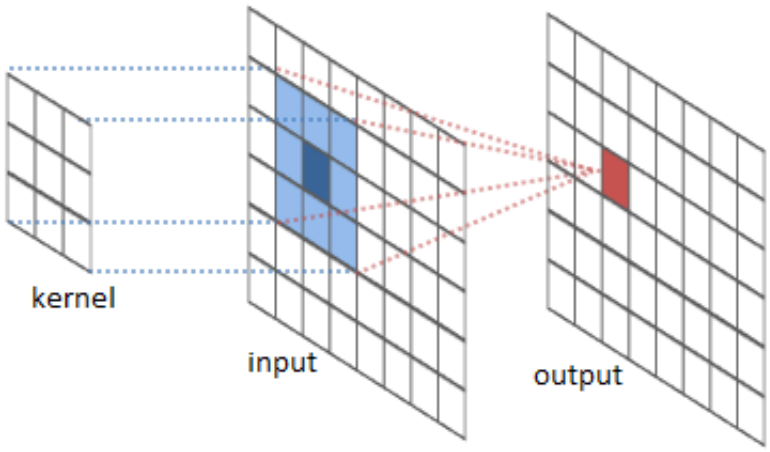
For an example, see the code under programming exercises. A common use case is when threads need to read from an array all threads have written to.

Convolutional Neural Networks (CNNs)

When implementing Convolutional Neural Networks (CNN) in CUDA one may use either standard convolution techniques or matrix multiplication. Explain how these two approaches work and the benefits and drawbacks with the two approaches.

Standard convolution

For each value in the input matrix, the kernel is used to calculate the corresponding value (sum of products) in the output matrix. Think lab 3 (image blurring).



Matrix multiplication

The convolution matrix / kernel is multiplied to a vector representation of the image on its left side. This yields a new matrix - a representation of the output image.

$$\underbrace{\mathbf{C}}_{\text{convolution matrix}} * \underbrace{\overline{n}}_{\text{image vector}} = \underbrace{\mathbf{P}}_{\text{processed image}}$$

Work-efficiency

A parallel algorithm can be work-efficient or not. The parallel algorithm is compared to the best performing sequential algorithm. Using the ordo notation, the parallel algorithm is said to be work-efficient if it requires as many operations as the sequential algorithm.

For example, if the sequential algorithm is considered to be $\mathcal{O}(n^2)$ and the parallel algorithm is found to be of $\mathcal{O}(n)$, the parallel algorithm can be said to be work-efficent.

Example (cumulative sum):

```
# Sequential: O(n)
int inputs[5] = {1, 2, 3, 4, 5};
int sums[5] = {0}
sums[0] = inputs[0];
for (int i = 1; i < 5; i++)
  sums[i] = inputs[i] + sums[i - 1]
```

MPI

MPI uses many processes distributed in a cluster. Each process computes part of the output and can communicate with eachother. Processes can synchronize.

MPI and CUDA

Give an example of an application that uses a combination of MPI and CUDA. Explain how MPI and CUDA are used to provide parallelism on different levels. Also, explain how CUDA streams could be used to improve the performance of such programs.

An example use cases is simulation of wave propagation.

MPI has support for message passing, something that is cumbersome on GPUs. MPI is also clustered across physical computers and can scale to large networks of computers. MPI has a high overhead.

CUDA has incredibly high parallelization but is not easily distributed across a network of computers.

The CUDA streams are used so that data can be transferred between the host and device when both devices are busy processing other data.

See also the previous section about CUDA streams.

Floating-point numbers

A floating point number consists of three parts. What are these three parts? What does normalization of floating point numbers mean.In what intervals do we not use normalization, and why?

A IEEE-754 floating-point numbers consist of a **sign**, a **mantissa** and an **exponent**.

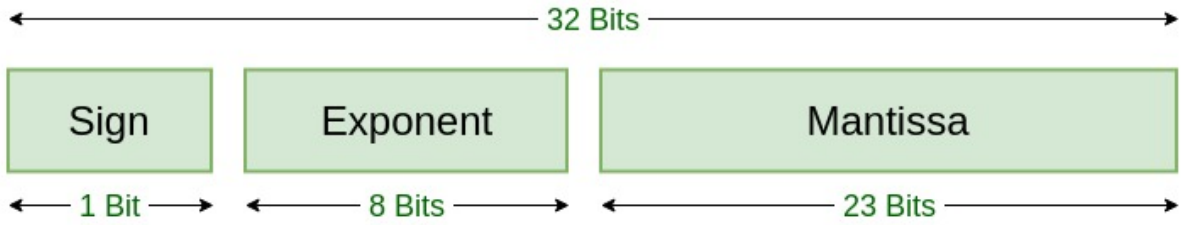
A floating point number is normalized when the integer part of its mantissa is 1, the fraction part may be anything . A special case is values around zero, since the requirement of the mantissa being 1 would not be able to be fulfilled.

Example of a non-normalized value:

$$13.25 \Rightarrow$$
$$\underbrace{1101}_{\text{integer part}} . \underbrace{01}_{\text{fraction part}} * (2^0)$$

Example of a normalized value:

$$13.25 \Rightarrow$$
$$1.10101 * (2^3)$$



Single Precision
IEEE 754 Floating-Point Standard