Notes on Programming Massively Parallel Processors

This is a documentation of my notes while studying from Kirk and Hwu's book on this topic.

1 Introduction to Data Parallelism and CUDA C

I have two hardware resources at my disposal - the *host*, which is the CPU, and the *device*, my GPU. A traditional C program is a CUDA program that contains only host code. Code meant for the device is marked with special CUDA keywords that are recognized by the `nvcc` compiler.

Thread Hierarchy

Data parallel functions in device code are called *kernels*. When a kernel function is called, it is executed by a large number of threads on the device. All the threads generated by a kernel launch constitute a *grid*. While the grid does its thing, the program can continue executing host code in parallel.

During kernel launch, the grid can be organized into blocks for the purpose of indexing threads. So, each block can be identified using up to three indices (depending on execution parameters). Thread block dimensionality can be accessed using `blockDim`.

Each thread in a grid has an associated `threadIdx`, which is a 3-component struct. So, threads can be identified using up to three-dimensional indices. So, for a block of size (N_x, N_y), the ID of a thread of index (i, j) is $(i + jN_x)$

A simple code for matrix addition using one block of threads is as follows.

```
__global__ void MatAdd(
        float A[n][n], float B[n][n], float C[n][n]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation with one block N^2 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(n, n);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Kernel functions are declared using the `__global__` keyword before the regular C-style function declaration. Their calls are also 'decorated' using the *execution configuration syntax* <<< ... >>>. This is a pair of number of threads per block and number of blocks per grid, which can be either `int` or `dim3`.

The most obvious example of data-parallel execution is vector addition. How does one go about doing that in CUDA? consider the following 'pseudo code'

```
#include <cuda.h>
...
void vecAdd(float *A, float* B, float* C, int n) {
    int size = n * sizeof(float);

    // 1
    // allocate device memory for A, B and C
    // copy A and B to device/global memory

    // 2
    // kernel launch code - to have the device
    // perform the actual vector addition

    // 3
    // copy C from device memory
    // free device vectors
}
```

Note that the device and host memory are separate - I am assuming that they are also separate virtually. Upon some looking up online I found a nice answer on stackoverflow. Essentially, this is a fast moving field and what I write here today may not hold on tomorrow's heterogeneous computing systems.

Memory Allocation and Data Movement

Global/device memory is allocated using the function `cudaMalloc()`. From `man cudamalloc`

```
__cudart_builtin__ cudaError_t cudaMalloc (
        void** devPtr, size_t size)
```

The first argument is the the address of a pointer variable that will be set to point to the allocated object (casted to `void **`). The address of the allocated memory is written to this pointer variable. I do not know how exactly the global memory is virtualized, so, I will try not to do anything funny with it. The host code passes this pointer value to the kernels that need to access the global memory.

One of the reasons for going for this two argument malloc is so that the return value can be used for error handling (as can be seen in the function call). You can choose to discard the return values and treat it like a `void` return type at your own risk I guess.

Here's a demo usage

```
float *d_A;
int size = n * sizeof(float);
cudaMalloc((void**)&d_A, size);
...
cudaFree(d_A);
```

So, this is also where I learned that `sizeof` in C is a compile-time operator and it seems to be very unique in this regard. Notice how in the above listing `float` has been passed as an argument to sizeof. The keyword `float` does not compile down to some object that would have some binary representation that can be stored in the registers and caches of your CPU. The assembly code generated uses hardcoded values from what I am guessing are lookup tables used by the compiler.

Once device memory has been allocated, data objects can be moved to the device memory using `cudaMemcpy()`. So, keeping in mind that pointers to device memory locations are different for pointers to host memory locations, the following is the memory allocation and movement part of `vecAdd`.

```
void vecAdd(float *A, float *B, float* C, int n) {
    int size = n * sizeof(float);  // bytes
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudamemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // kernel invocation
    ...

    // retrieving result of computation from device to host
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Kernel Functions and Threading

CUDA programming is SPMD, so we are not limited to the same pipeline for all the data. So, we can go beyond vector addition.

Anyway. As mentioned before, a kernel launch spawns a grid of threads. There is another layer to this hierarchy - blocks. The grid is uniformly divided up into blocks of (up to 1024) threads. The number of threads spawned in each block is defined by the host code at the function call (keep it a multiple of 32). The same kernel can be launched with a variable number of threads at different locations in a program.

Each block has an identifier `blockIdx`. There are `blockDim` threads per block. Each thread in a block has an identifier `threadIdx`. So, data entry i of an array can be identified as

```
i = blockId.x*blockDim.x + threadIdx.x
```

The kernel function definition looks like this.

```
// each thread performs one scalar addition
__global__
void vecAddKernel(float *A, float *B, float *C, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n) C[i] = A[i] + B[i];
}
```

The `.x` implies that there would be `.y` and `.z` too. Note that the block and thread identifiers are private to each thread, which means that the variables `i` is also private to each thread. `i` is called an *automatic variable.*

An `i < n` conditonal is required because the vector may not occupy the entirety of the last block.

The keyword list (executed on the <- only callable from the):

- `__device__ float DeviceFunc()`: device <- device, can only be called from a kernel function or another device function
- `__global__ void KernelFunc()`: device <- host
- `__host__ float HostFunc()`: host <- host

Now that the kernel function has been defined, it can be called.

```
int vecAdd(float* A, float *B, float *C, int n) {
    ...
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

The final code, `vecadd.cu` looks like so

```
#include <cuda.h>
#include "vecadd.h"

__global__
void vecAddKernel(float* A, float *B, float* C, int n) {
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    if (i < n)
        C[i] = A[i] + B[i];
}

void vecAdd(float *A, float *B, float *C, int n) {
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Listing 1: Vector addition kernel function and kernel launch function

Epic. Note that `nvcc` requires a `main()` function and you cannot compile to relocatables individually. Or so it would seem, I have not yet looked into all the flags etc that can be passed to `nvcc`.

The following is a matrix addition code that does one scalar addition per thread.

```
#include <cuda.h>
#include "matrixAdd.h"

// TODO: remove boilerplate using preprocessor macros
__global__
void
matrixAddKernel_elementwise(
        float *A, float *B, float *C, int n) {
    // A, B, and C are device memory pointers
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    int j = blockDim.y*blockIdx.y + threadIdx.y;
    if (i < n && j < n) {
        int k = n*i + j;
        C[k] = A[k] + B[k];
    }
}


void
matrixAdd(float *A, float *B,
        float *C, int n) {
    int size = n * n * sizeof(float);
    float *d_A, *d_B, *d_C;  // device memory pointers
    const dim3 threadsPerBlock(16, 16);  // 256 threads per block
    const dim3 numBlocks(ceil(n/threadsPerBlock.x),
    ↪  ceil(n/threadsPerBlock.y));

    cudaMalloc((void**) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    matrixAddKernel_elementwise<<<numBlocks, threadsPerBlock>>>(
            d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B), cudaFree(d_C);
}


__global__
void
matrixAddKernel_roww(float *A, float *B, float *C, int n) {
    // each thread performs matrix addition for one row
    int row = blockDim.x*blockIdx.x + threadIdx.x;
    if (row < n) {
        for (int i = 0; i < n; i++) {
            int ind = n*row + i;
            C[ind] = A[ind] + B[ind];
        }
    }
}


void
matrixAdd_roww(float *A, float *B, float *C, int n) {
    float *d_A, *d_B, *d_C;
    const int threadsPerBlock = 128;
    const int numBlocks = (int) ceil(((float)n)/threadsPerBlock);
    const int size = n*n;

    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    matrixAddKernel_roww<<<numBlocks, threadsPerBlock>>>(
            d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B), cudaFree(d_C);
}


__global__
void
matrixAddKernel_colw(float *A, float *B, float *C, int n) {
    int col = threadIdx.x + blockDim.x*blockIdx.x;
    if (col < n) {
        for (int i = 0; i < n; i ++) {
            int index = n*i + col;
            C[index] = A[index] + B[index];
        }
    }
}


void
matrixAdd_colw(float *A, float *B, float *C, int n) {
    float *d_A, *d_B, *d_C;
    const int threadsPerBlock = 128;
    const int numBlocks = (int) ceil(((float)n)/threadsPerBlock);
    const int size = n*n;

    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    matrixAddKernel_colw<<<numBlocks, threadsPerBlock>>>(
            d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B), cudaFree(d_C);
}
```

Listing 2: Kernel and host functions for performing matrix addition

Data-Parallel Execution Model

The execution is SPMD, with each thread using unique coordinates to access some region of the data-structure. A grid is a 3D-array of blocks and a block is a 3D array of threads.

Each block has a 3-vector `blockIdx`, and each thread has a 3-vector `threadIdx`, the components having identifiers `x`, `y`, `z`. dimensions of each block and each grid can be specified using variables of type `dim3`. If scalar dimension specifiers are used (e.g. `int`), then the `x` dimension is set to the scalar value while the `y` and `z` are set to `1`. A grid can have higher dimensionality then its blocks.

It might be a good practice to assign indices such that the largest of the dimensions comes first, e.g.

```
dim3 dimBlock(2, 2, 1);
dim3 dimGrid(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

This works better when mapping thread coordinates into data indices in accessing multidimensional arrays.

NVIDIA GPUs in 2012 could support up to 1024 threads per block with flexibility in how one can index them.

Mapping Threads To Multidimensional Data

The programmer needs to think about the dimensionality of the grid and blocks based on the nature of the data in consideration. E.g. an image is a 2D grid of values. If the image is greyscale, then we can do with a representation that takes a scalar value per pixel.

So, if the resolution of our image is 76×62, we could create a grid using 16×16 blocks, and let the grid be 5×4 blocks. 16×16 is a typical choice of block dimensions for 2D data structures. In this example, we are allocating 80×64 threads to process 76×62 pixels.

To avoid wasteful computations, there should be a conditional guard so that spawned threads in the grid that do not have pixel values to work with do not do any computations, like was done in the vector addition example, wherein four 256-thread blocks were spawned to deal with a vector of length 1000.

If you have been provided with parameters for a 2D array, e.g. the dimensions $m \times n$ for the image, then you can use something like this two spawn a grid for processing the image.

```
dim3 dimBlock(ceil(n/16.0), ceil(m/16.0));
dim3 dimGrid(16, 16, 1);
pictureKernel<<<dimGrid, dimBlock>>>(d_Pin, d_Pout, n, m);
```

So, here's the important bit. We have a global memory pointer `d_Pin` for our input data. As of 2012, CUDA C is based on ANSI C, which requires the number of rows and columns in 2D arrays to be known at compile time, which spells doom for most of our dynamically allocated datastructures. So, the programmer has to explicitly flatten the tensor to an equivalent 1D array.

The most up to date documents on NVIDIA are on C++. The C documents are 2010-ish old.The CUDA 11.0 guides on programming and best practices are exclusive to C++.

Checking C99 compatibility would require some testing. For now, all I need to know is that C arrays are stored in row-major order.

The following listing shows an example of a CUDA code for matrix multiplication for square matrices

```
#include <cuda.h>
#include "matrixMul.h"

__global__ void matrixMulSquareKernel(float *d_M, float *d_N,
        float *d_P, int width) {
    // P_{ij} = M_ikN_kj
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if ((row < width && col < width)) {
        int k;
        int P_ind = row * width + col;
        d_P[P_ind] = 0.0;
        for (k = 0; k < width; k++) {
                d_P[P_ind] += d_M[row*width + k]*d_N[k*width + col];
            }
        }
}

void matrixMulSquare(float *M, float *N, float *P, int width) {
    const dim3 dimBlock(16, 16);
    const int numBlocks = ceil(width/16.0);
    const dim3 dimGrid(numBlocks, numBlocks);
    float *d_M, *d_N, *d_P;

    int sz = width*width * sizeof(float);

    cudaMalloc((void **) &d_M, sz);
    cudaMalloc((void **) &d_N, sz);
    cudaMalloc((void **) &d_P, sz);

    cudaMemcpy(d_M, M, sz, cudaMemcpyHostToDevice);
    cudaMemcpy(d_N, M, sz, cudaMemcpyHostToDevice);

    matrixMulSquareKernel<<<dimGrid, dimBlock>>>(d_M,
            d_N, d_P, width);

    cudaMemcpy(P, d_P, sz, cudaMemcpyDeviceToHost);
    cudaFree(d_M); cudaFree(d_N); cudaFree(d_P);
}

void matrix_vectorMulKernel(float *A, float *v, float *p, int n)
↪   {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index < n) {
        p[index] = 0.0;
        for (int i = 0; i < n; i++) {
            int matindex = n*index + i;
            p[index] += A[matindex]*v[i];
        }
    }
}

void matrix_vectorMul(float *A, float *v, float *p, int n) {
    const int threadsPerBlock = 16;
    const int numBlocks = n/threadsPerBlock + 1;
    int size = n*n;

    float *d_A, *d_v, *d_p;

    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_v, n);
    cudaMalloc((void **) &d_p, n);

    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_v, v, n, cudaMemcpyHostToDevice);

    matrix_vectorMulKernel<<<numBlocks, threadsPerBlock>>>(
            d_A, d_v, d_p, n);

    cudaMemcpy(p, d_p, n, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_v); cudaFree(d_p);
}
```

Listing 3: Multiplication of square matrices

Synchronization and Transparent Scalability

In general one has to coordinate the execution of threads. CUDA allows threads in the same block to coordinate using a barrier synchronization function `__syncthreads()`. All threads in the calling block will be held until all threads reach the same location in the program. I see potential deadlocks.

At the very least using this particular barrier synchronization would allow different blocks to execute independent of each other.

Thread Scheduling and Latency Tolerance

Thread scheduling depend on specific hardware implementations, A warp is a unit of thread scheduling, typically 32 threads, but is not defined in the CUDA specification. A streaming multiprocessor executes all threads in a WARP in a SIMD fashion - all threads in a warp will have the same execution timing.

In general, there are fewer streaming processors (SP) per SM than the number of threads assigned to each SM; a scheduler is involved in the execution of all the warps assigned to an SM. By having more warps per SM than what the underlying hardware can execute at a time, CUDA processors efficiently execute long latency operations such as global memory accesses - context switching. Another resident warp that is not scheduled is then scheduled and the previously scheduled warp, that is waiting for the completion of a long latency operation is queued.

Examples of long latency operations include pipelined FP arithmetic, branch instructions etc. *Note that warp scheduling has zero-overhead when switching to another ready warp.*

SIMT and Warp-Level Primitives

I came across a nice blog post by NVIDIA on warp-level primitives. The question that I had been trying to address was "is it safe to remove `__syncthreads()` when the block size is the same as the device warp size?". The answer is no. There can be divergent control flow between threads in the same warp. The simplest example is when checking boundary conditions when working with tensors/tensor-like data structures.

While GPU computing is usually defined as SIMD (Single Instruction Multiple Thread), the classification SIMD in the context of Flynn's taxonomy is usually reserved for vector architecture, i.e. a scalar thread issues vector instructions, and the exact same instruction is applied across many data elements.

SIMT (Single Instruction Multiple Thread) is a more appropriate classification for GPUs, even at the level of a single warp. Each thread has access to its own registers and can load and store from divergent addresses and follow different control paths.

It is possible to achieve even higher performance by using warp-level primitives and exploiting the flexibility provided by this SIMT architecture.

A consequence of this is that we can have cooperative groups collectives, e.g. accumulating the sum of values in all threads into a single thread (analogous to `MPI_Reduce`).

```
#define FULL_MASK 0xffffffff
for (int offset = 0x10; offset > 0; offset /= 2)
    val += __shfl_down_sync(FULL_MASK, val, offset);
```

The above code performs a tree-reduction to compute the sum of the `val` variable held by each thread in a warp. See figure 1

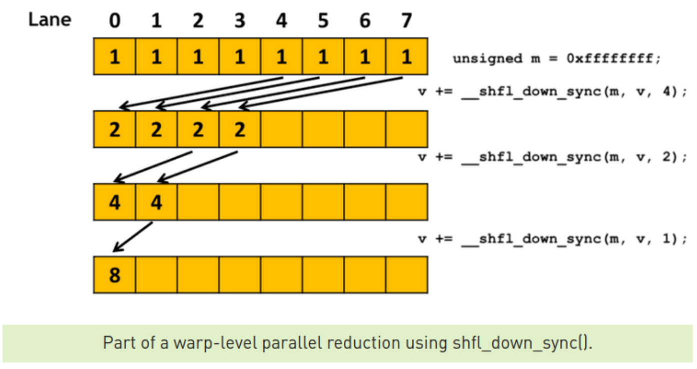
Lane
0 1 2 3 4 5 6 7
1 1 1 1 1 1 1 1
unsigned m = 0xffffffff;
v += __shfl_down_sync(m, v, 4);
2 2 2 2
v += __shfl_down_sync(m, v, 2);
4 4
v += __shfl_down_sync(m, v, 1);
8
Part of a warp-level parallel reduction using shfl_down_sync().

Figure 1: Warp-level parallel reduction (taken from NVIDIA's blog)

(TODO: document some warp-level primitives)

A Few Words About GPU Architecture

Essentially,

Table 1: CUDA Software-Hardware mapping

Software		**Hardware**
CUDA Thread	executes on/as	CUDA Core/ SIMD Core
CUDA Block	executes on	Streaming Multiprocessor
GRID/kernel	executes on	GPU Device

CUDA threads are more lightweight and provide faster context switches than CPU threads because of larger register size and hardware-based scheduler. The

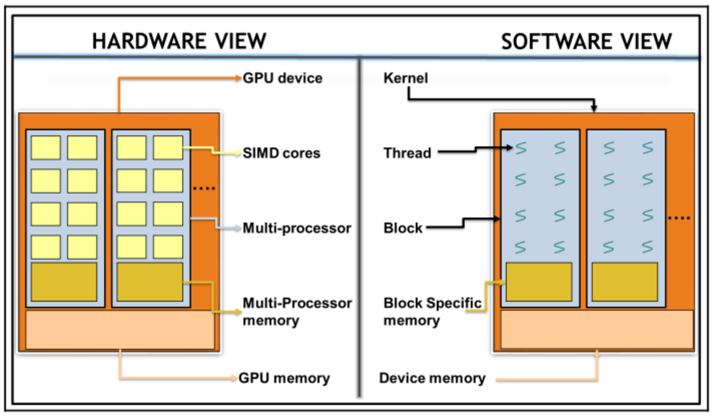

Figure 2: CUDA Hardware and Programming Model Overview

thread context is stored in the registers, as opposed to being on the cache in CPUs.

CUDA blocks execute on a single streaming multiprocessor (SM). All the threads of a block can only execute on one SM. In general, a GPU may have multiple SMs. The CUDA programming model allows for communication between threads in the same block; threads belonging to different blocks cannot communicate/synchronize.

CUDA provides a `cudaDeviceProp` data type along with associated functions that can be used to get the properties of the device(s) that can communicate with host.

Error reporting in CUDA

Errors are managed by the host code. Most CUDA functions return an enumation type `cudaError_t`, with `cudaSuccess` being code 0. Here is a typical example.

```
cudaError_t e;
e = cudaMemcpy(...);
if (e)
    fprintf(stderr, "Error: %s\n", cudaGetErrorString(err));
```

Kernel launches are always `void` return-type. So, there is also a very handy function `cudaGetLastError()`

```
MyKernel<<<...>>> (...);
cudaDeviceSynchronize();
e = cudaGetLastError();
```

Error checking is important in CUDA because the host will continue executing even if a `__global__` function (kernel) has crashed.

CUDA Memory Management

In most intensive applications, memory-related operations present a bottleneck (same goes for CPU programs too I guess, they call it the von Neumann bottleneck). Typically, your memory bandwidth is going to be smaller than your FLOPS, and in the worst case your FLOPS count will be capped by the memory bandwidth. This should not happen too much because GPU is a latency-hiding architecture - large latency memory accesses are hidden by high throughput execution.

Which is why here also we have caches, along with larger registers. Take a look at figure 3. Each memory bandwidth is orders of magnitude different.

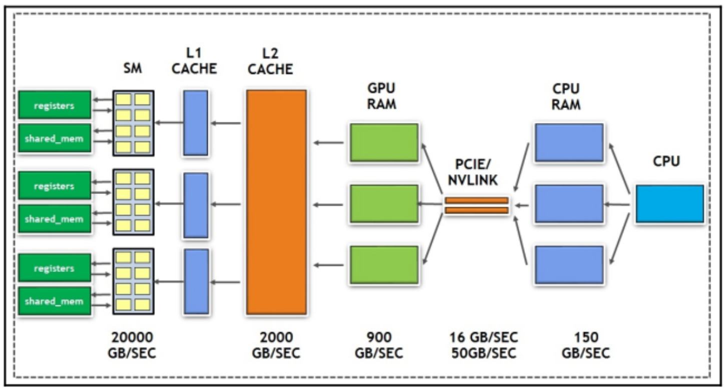

Figure 3: Memory pathways in a typical host-device machine with memory bandwidths(taken from the book 'Learn CUDA Programming' by Packtpub)

The memory hierarchy in a CUDA device (and in any typical GPU) is a bit more involved than that of a PC host.

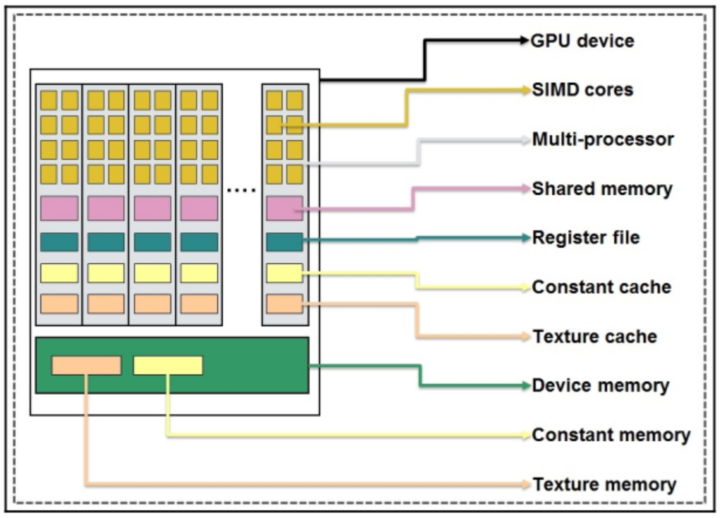

Figure 4: Different types of memory present in a typical GPU

One can optimally utilize different types of GPU memories.

- Global memory
- Shared memory
- Read-only data/cache
- Pinned memory
- Unified memory

Profiling Tools

The CUDA Toolkit have the following main tools

- nvcc - the compiler
- cuobjdump - similar to objdump
- nvdisasm - disassembler
- nvprune - pruning tool
- nsight (???)
- nvvp - visual profiler
- nvprof - command line profiler
- cuda-memcheck

`nvvp` is a full-fledged GUI application, complete with the Autodesk-like splash screen :puke-emoji:

Go and read the man pages.

. . . TBD

CUDA Memories

The previous implementations that use only global memory accesses to move and access data achieve only a fraction of the potential of the GPU. DRAMs typcically have access latencies in some hundreds of clock cycles and finite access bandwidth. So, scheduling can hide global memory access latency only up to a certain point.

Consider for example the matrix multiplication kernel. The workhorse of the kernel was the `for` loop

```
for (int k = 0; k < width; k++)
    Pvalue += d_M[row*width + k]*d_N[k*width + col]
```

There are two global memory accesses per iteration of the loop along with two floating point addtions. Compute:Global-memory-access ratio (CGMA) is 1:1.

For a memory bandwidth of 200 GB/s, we can fetch $200/4 = 50$ floats per second, and consequently perform at 50 GFLOPS. But, the peak performance of cards with such specs is around 1500 GFLOPS. Yikes. We need a CGMA of 30 to get there, with modern GPUs demanding even higher CGMA for reaching full potential.

CUDA device memory types

- Registers (on-chip, allocated to individual threads)
- Shared Memory (on-chip, allocated to individual blocks)
- Texture/Constant/Read-only memory
- . . . (newer architectures)
- Device code can
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read per-grid constant memory
- Host code can
  - Transfer data to/from per grid global and constant memories.

The global memory is analogous to the memory in the von Neumann model, i.e. implemented off the chip. The registers are analogous to the register files in the von Neumann architecture. These have drastically shorter latencies and higher agregate access bandwidth.

Shared Memory

Allocated using the `__shared__` memory space identifier. It is typically used as a scratchpad memory (or software managed cache) to minimize global memory accesses from a CUDA block.

The global memory implementation of a matrix multiplication kernel has a CGMA of 1.0. A shared memory implementation of matrix multiplication is going to be much faster. it is also called *tiled matrix multiplication* as the input and output matrices are partitioned like tiles on a floor and allocated to different blocks, with each block having its own shared memory.

Using shared memory is one strategy to reduce global memory traffic.