GPU Programming with CUDA

Prof. Dr. V. Stahl
Heilbronn University of Applied Sciences

November 29, 2025

Contents

1 First Steps

1.1 How to do Experiments

In case you don't have a computer with a CUDA GPU, a native CUDA driver and CUDA toolkit, you can still do hands on experiments with CUDA using Google Colab for free, see [5].

1.2 First Example

Let us begin with a simple CUDA program.

```
#include <stdio.h>

// This function is executed by every thread on the GPU (device).
__global__ void mykernel()
{
  // Every thread prints its block and thread ID.
  printf("block/thread id %d/%d\n",blockIdx.x,threadIdx.x);
}


// The main function runs on the CPU (host).
int main(void)
{
  // Start 2 blocks with 3 threads on GPU.
  mykernel<<<2,3>>>();

  // Wait for GPU to finish.
  cudaDeviceSynchronize();

  return 0;
}
```

The program is mostly C++ except for the line `mykernel<<<2,3>>>`. It has therefore to be translated with a special NVIDA compiler called `nvcc`.

Functions which run in threads on the GPU are called "kernels" and have to be declared with `__global__`.

The program starts 6 parallel threads which are organized in two blocks with three threads per block. The set of all blocks is called grid. Each thread prints the id of the block it is in and the id of the thread within its block.

1.3 Parallel Vector Addition

The following program adds two vectors in parallel on the GPU and verifies the result on the CPU.

```
#include <stdio.h>

// Kernel function to add the elements of two arrays.
__global__ void add(int n, float* x, float* y, float* z)
{
  int numBlocks = gridDim.x;           // Number of blocks.
  int numThreadsPerBlock = blockDim.x; // Number of threads in each block.
  int blockIndex = blockIdx.x;         // Index of block in grid.
  int threadIndex = threadIdx.x;       // Index of thread in block.

  int threadId = blockIndex * numThreadsPerBlock + threadIndex;
  int numthreads = numThreadsPerBlock * numBlocks;

  for(int i=threadId; i<n; i+=numthreads)
    z[i] = x[i] + y[i];
}

int main(void)
{
  int n = 1<<20;
  int nbytes = n*sizeof(float);
  float *x, *y, *z;

  // Allocate unified memory - accessible from both CPU and GPU.
  cudaMallocManaged(&x, nbytes);
  cudaMallocManaged(&y, nbytes);
  cudaMallocManaged(&z, nbytes);

  // Initialize x and y arrays on the host.
  for(int i=0; i<n; i++)
  {
    x[i] = ...;
    y[i] = ...;
  }
  add<<<20, 32>>>(n,x,y,z);  // 20 blocks, 32 threads per block

  // Wait for GPU to finish.
  cudaDeviceSynchronize();

  // Compare addition on GPU with addition on CPU.
  float maxerr = 0.0f;
  for (int i=0; i<n; i++)
    maxerr = fmax(maxerr, x[i]+y[i]-z[i]);
  printf("Max error: %f\n",maxerr);

  // Free memory.
  cudaFree(x); cudaFree(y); cudaFree(z);
  return 0;
}
```

The program starts $20 \times 32 = 640$ threads to add a vector with $2^{20} \approx 10^6$ components, such that each thread adds about 1634 numbers. The work is distributed such that thread 0 adds components $0, 32, 64, \ldots$, thread 1 adds components $1, 33, 65, \ldots$ and so on. For this purpose each thread calculates its id `threadId` and the total number of threads `numthreads`.

Memory has to be allocated to store the input vectors x and y and the result z. The problem is that CPU and GPU use separate memory. The memory of the CPU is on the motherboard whereas the memory of the GPU is on the graphics card. Therefore a copy of x, y, z is needed in both memories and data has to be transferred at the right time in the proper direction. By allocating memory with `cudaMallocManaged()` this is done transparently. Memory allocated in this way has to be freed with `cudaFree()`.

If you want to program memory allocation and data transfer between host and device explicitly, it works as follows:

```
float * xhost;   // Address in host memory.
float * xdevice; // Address in device memory.

// Allocate memory on host and on device.
xhost = (float*)malloc(nbytes);
cudaMalloc(&xdevice, nbytes);

// Transfer data from host to device.
cudaMemcpy(xdevice, xhost, nbytes, cudaMemcpyHostToDevice));

// Transfer data from device to host.
cudaMemcpy(xhost, xdevice, nbytes, cudaMemcpyDeviceToHost));

// Free memory.
free(xhost);
cudaFree(xdevice);
```

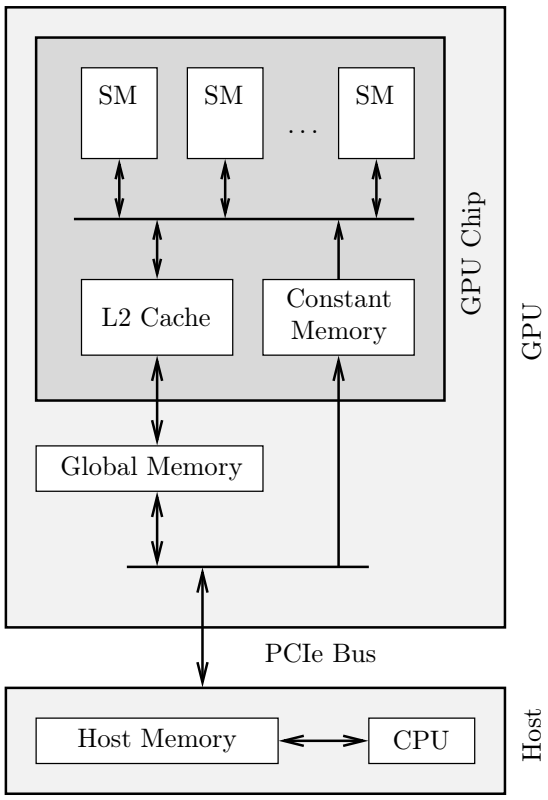
2 Architecture of a GPU

In order to understand why threads are grouped into blocks, how they communicate and how to achieve good performance, a basic understanding of the architecture of a GPU is necessary.

2.1 Processors

The main components of a graphics processing unit (GPU), also called device, is the GPU chip and memory chips.

- **Graphics Processor (GPU chip).** The GPU chip consists of several streaming multiprocessors (SM), L2 cache memory and constant memory.



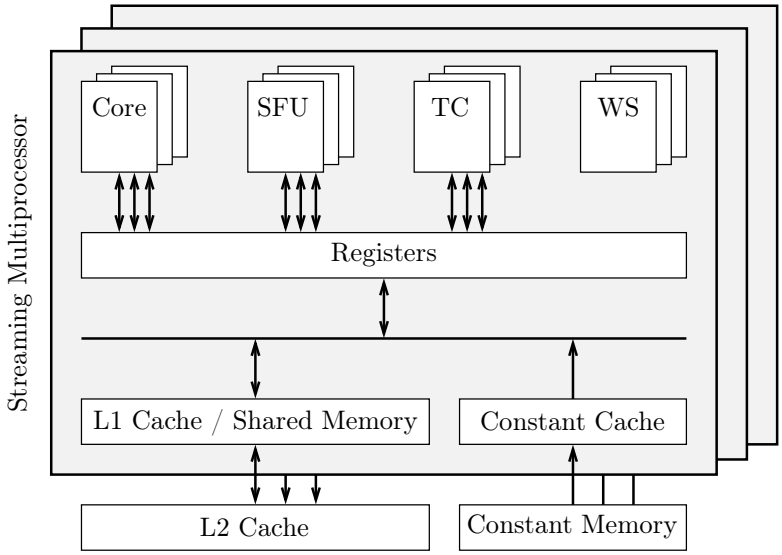
  Let's take the Nvidia GeForce RTX 4060 as an example. It was introduced on June 29, 2023 with a starting price of 299$. It has 8GB off chip global memory, 24 MB L2 cache and 64kB constant memory. The bus width between off chip global memory and L2 cache is 128 bits with a peak bandwidth of 272 GB/s. Its GPU chip is a AD107 (Ada Lovelace microarchitecture) and has 24 SMs.

- **Streaming Multiprocessor (SM).**
  An SM consists of several (usually 128) CUDA cores. It has special memory which can be split into L1 cache and shared memory and is accessible to all cores. The shared memory can be used for communication and synchronization among threads running in a block on the SM. Thread blocks are explained in detail in Section 3, for the moment it suffices to know that a block is a logical group of

threads. While the cache controller decides what data is stored and replaced in the L1 cache, the programmer has explicit control over shared memory. Further each SM has a small cache to access constant memory.

An SM has several special function units (SFU) for functions like, sin, cos, log, tensor cores (TC) for fast matrix operations and warp schedulers (WS).

In order to schedule the parallel threads running on an SM it has several (typically 4) warp schedulers (WS).



On the Nvidia GeForce RTX 4060 each SM has 128 cores, 4 SFU, 4 WS, 100 kB of shared memory /L1 cache, 8kB constant cache and 56 536 registers with 32bit.

- **CUDA Core.**

  A core consists usually of a single precision FP32 ALU and an an INT32 ALU. The ALUs have a pipelined multiplier and adder. These can be configured such that in each cycle a fused addition-multiplication can be carried out, which is the basic operation for computing dot products.

  On the Nvidia GeForce RTX 4060 some of the hardware is shared between cores. Therefore it is either possible to execute 128 FP32 operations or 64 FP32 and 64 INT32 operations per cycle on an SM.

  Double precision floating point arithmetic is also possible, but slower by factor 64 compared to single precision.

With a base/boost clock frequency of 1.83/2.46 GHz and $24 \times 128 = 3\,072$ cores, the RTX 4060 has a theoretical peak FP32 performance of 15 TFLOPS (multiply-add counts as two operations) but only 236 GFLOPS with FP64.

2.2 Memory

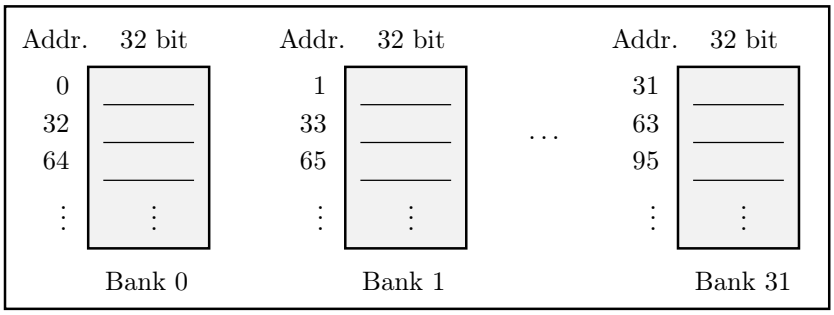
There are several kinds of memory on a GPU:

- **Global memory.** This is off-chip memory on the graphics card, which is shared by all SMs. Global memory is accessed through an on-chip L2 cache.

- **Constant memory.** Like global memory this memory can be accessed by all SMs and comes with a local cache on each SM. It can be written only by the host, but not by the GPU. The advantage of constant memory is that it can reduce memory latency significantly, especially because of the cache on each SM.
- **L1 cache/shared memory** on each SM. This is a physical memory device which can be divided logically into two parts by the programmer:
  - **L1 cache.** This is a shared resource accessible by *all threads* running on the SM. If different blocks running on an SM access the same data in global memory, it is copied only once to the L1 cache of that SM. A cache controller (hardware) manages loading data from global memory into L1 cache. This controller decides where to store data within the cache, and determines which cache lines to replace when the cache is full. The primary goal is to minimize cache misses and improve memory access performance.
  - **Shared memory.** A local variable in a kernel declared with modifier `__shared__` is shared among all threads of a block. It can therefore be used for communication between threads of the *same block*. In contrast to L1 cache, shared memory is software managed, which means that the code of a kernel has to make explicit, which data is stored in shared memory. Shared memory is organized into 32 independent banks. This banking allows parallel threads to access consecutive word addresses efficiently with minimal conflicts, which maximizes memory bandwidth and reduces access latency.

    While threads can communicate via global memory as well, it is significantly slower than shared memory due to higher latency and lower bandwidth. Further, global memory is shared between all SMs of the GPU whereas shared memory is local to each SM.

Shared Memory



- **Unified memory.** CPU and GPU have separate address spaces and data has to be copied between main memory of the CPU and global memory of the GPU with explicit function calls. Unified (or managed) memory is a common, virtual adress space for CPU and GPU. Data transfers are done as needed by the memory management system in a transparent way.

Latency differs significantly between the above mentioned memories and is important to take into account when developing software for GPUs. Typical figures are as follows:

Registers	1 cycle
Constant cache	$20 - 40$ cycles
L1 cache	$20 - 40$ cycles
L2 cache	$200 - 300$ cycles
Constant memory	$200 - 300$ cycles
Global memory	$400 - 800$ cycles

In order to avoid idle times while waiting for data traveling up the memory hierarchy, GPUs can switch between threads (more precisely groups of threads called warps) with no delay. Thus, while one thread waits, another one can take over. This so called "latency hiding" works well as long as there are sufficiently many active threads. A GPU has no other layers of cache apart from L1 and L2 and relies heavily on latency hiding to gain efficiency. In contrast, CPUs use typically 3 layers of cache to reduce latency.

3 Kernels, Threads and Thread Blocks

3.1 Kernels

A kernel is a C++ function, which is executed on the GPU by many threads in parallel. It is declared with

`__global__ void mykernel(` *argument list* `)`

and called from a function running on the host with

`mykernel<<<`*gridsize, blocksize*`>>>(` *argument list* `);`

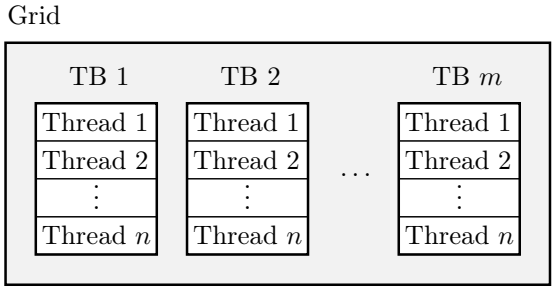
As this is not C++ syntax, a GPU program has to be compiled with a special compiler called `nvcc`, which is a frontend to the GNU C++ compiler on Linux. The meaning of gridsize and blocksize is described in Section 3.2.

The modifier `__global__` in a function declaration means that the function is called from the host, but runs on the device. A modifier `__device__` means that the function can be called only from the device and runs on the device. There is a third (optional) parameter `__host__` for functions which are called by the host and run on the host. The modifiers can be combined to tell the compiler to generate code for both host and device.

3.2 Thread Blocks

Threads are grouped into thread blocks (TB). All threads within a block execute the same kernel, run on the same SM and can not be distributed over several SMs. The reason is that threads within a block can communicate and synchronize using the shared memory of the SM they are running on. An SM can execute several thread blocks simultaneously. Thread blocks can not communicate with each other.

Threads blocks are grouped into a grid. All blocks in a grid contain the same number of threads.



The correspondence between software and hardware can be summarized as follows:

thread	executes a	kernel	(C++ function)
block	runs on a	SM	(hardware)
grid	runs on a	GPU	(hardware)

When a thread executes a kernel, it can figure out the gridsize, the blocksize, the id of its block and its id within the block. This information can be used to obtain a unique id of the thread:

```
__global__ void mykernel(...)
{
  int gridsize  = gridDim.x;
  int blocksize = blockDim.x;
  int blockid   = blockIdx.x;
  int threadid  = threadIdx.x;

  int myid = blocksize*blockid + threadid;
  ...
}
```

The reason for the x's is that threads and blocks can be organized in two- or three-dimensional arrays.

3.3 Warps

The threads within a block are grouped at runtime into fixed warps consisting of 32 threads. As a consequence every thread in a warp belongs to the same block and executes the same kernel code. If a the blocksize is not a multiple of 32, then a warp may contain less than 32 threads leaving some of its lanes empty.

In early Nvidia microarchitectures all threads in a warp executed the same instruction in every cycle (lockstep synchronous). If a branch occured, the two paths had to be executed sequentially. The threads which took one branch were executed while the others were masked out and vice versa. This policy is called single instruction multiple threads (SIMT) model. The difference to the single instruction multiple data (SIMD) scheme according to [4] is that the former allows thread specific branches.

Starting with the Volta microarchitecture in 2017, Nvidia introduced independent thread scheduling. This means that each thread in a warp can now step independently through instructions and has its own program counter. This improves utilization but breaks the old implicit lockstep assumption.

Usually there are 4 warp schedulers per SM. The task of a warp scheduler is to pick a warp which is ready to run and let it use the hardware of the SM. With 4 warp schedulers and 32 threads per warp, each of the 128 cores of the SM can be utilized. Some warp schedulers have a "dual-issue capability", which means that they can issue two independent instructions to a warp in the same cycle. For example, a scheduler could issue a floating-point operation and a memory operation simultaneously.

Different warps execute independently. Context switches between warps cost nothing because each warp has its own physical set of registers.

The choice of gridsize and blocksize depends on the underlying hardware and data dependendies. As a general rule, the blocksize should be a multiple of a warp. A good starting point is 128 or 256. With 128 threads per block a single block can occupy an SM entirely. In order to avoid idle times due to memory latencies, 256 or 512 is often a better choice. The gridsize is chosen such that enough threads are activated to occupy all cores. A limiting factor is that the resources of an SM like for example its shared memory have to be subdivided among the blocks assigned to this SM. If there are too many blocks, the shared memory of each block might run out and the blocks have to run sequentially.

4 Floating Point Arithmetic

According to the IEEE Standard 754 on binary floating point arithmetic [7] [8] the result of an addition or multiplication is always rounded to a representable floating point number. As a consequence, laws of arithmetic like

$$\begin{aligned}(x+y)z &= xz+yz\\ (x+y)+z &= x+(y+z)\end{aligned}$$

no longer hold for floating point numbers and optimizing code by rearranging the order of arithmetic operations in parallel algorithms may result in different results.

In many applications like dot products, a multiplication is followed by an addition. In a revision of the IEEE Standard 754 a fused multiply-add (FMA) operation was added. With the FMA operation we obtain

$$\text{round}(xy+z)$$

instead of

$$\text{round}(\text{round}(xy)+z).$$

Therefore the FMA operation gives more accurate results and is usually faster.

While GPUs use FMA [3] [11], ordinary CPUs don't by default. When comparing results of a GPU with a CPU one has to compile with flag `fmad` as in

```
nvcc --fmad false program.cu
```

to switch off FMA on the GPU.

A better choice is to enable FMA on the CPU in order to obtain results which are identical on CPU and GPU. The compiler flags for the GNU compiler are

```
nvcc -Xcompiler -mfma program.cu
```

see documentation on `nvcc` in [10].

More information on floating point arithmetic with CUDA can be found in [2].

5 Status of the GPU

The status of the GPU can be queried from the command line with `nvidia-smi` or at runtime from a CUDA program using the Nvidia Management Library (NVML).

5.1 Command line Tool `nvidia-smi`

The current status of the GPU can be listed with

```
nvidia-smi -q
```

If one is interested only in certain aspects like the clock frequency or the temperature and their limits, one can query with

```
nvidia-smi -q -d clock
nvidia-smi -q -d temperature
```

If one wants to know only the value of a certain variable like the current temperature or the clock frequency of the SMs, the query is

```
nvidia-smi --query-gpu=temperature.gpu --format=csv,noheader
nvidia-smi --query-gpu=clocks.current.sm --format=csv,noheader
```

A list of all available variables is obtained with

```
nvidia-smi --help-query-gpu
```

5.2 Status at Runtime with NVML

The current clock frequency of the SMs and memory as well as the temperature can be checked from a CUDA program with the NVML API:

```
#include <nvml.h>

nvmlDevice_t device;
unsigned int smclock, memclock, temp;

nvmlInit();
nvmlDeviceGetHandleByIndex(0, &device);

nvmlDeviceGetClockInfo(device, NVML_CLOCK_SM,  &smclock);
nvmlDeviceGetClockInfo(device, NVML_CLOCK_MEM, &memclock);
nvmlDeviceGetTemperature(device, NVML_TEMPERATURE_GPU, &temp);

printf("current SM clock: %u MHz", smclock);
printf("current memory clock: %u MHz, memclock);
printf("current temperature: %u\n", temp);
```

The program has to be linked with `libnvidia-ml.so` or using flag `-lnvidia-ml`:

```
nvcc program.cu -lnvidia-ml
```

Under linux the library is installed here:

```
/usr/lib/x86_64-linux-gnu/libnvidia-ml.so
```

6 Measuring Computing Time

Computing time can be measured at runtime using events or evaluated after completion of the program with a profiler. In both cases one has to take into account that the GPU clock frequency may vary at startup or be reduced to avoid overheating.

6.1 CUDA Events

The recommended method to measure computing time at runtime is the use of CUDA events. The usage is as follows:

```
cudaEvent_t event1, event2;
cudaEventCreate(&event1);
cudaEventCreate(&event2);

// Add event1 to instruction stream on GPU.
cudaEventRecord(event1);

// Add kernels to instruction stream on GPU.
mykernel<<<gridsize, blocksize>>>(...);

// Add event2 to instruction stream on GPU.
cudaEventRecord(event2);

// Host waits until event2 was processed by GPU.
cudaEventSynchronize(event2);

// Elapsed time is difference between the time stamps of the two events.
float milliseconds = 0.0;
cudaEventElapsedTime(&milliseconds, event1, event2);

// Cleanup
cudaEventDestroy(event1);
cudaEventDestroy(event2);
```

This measures the wall-clock computing time of the kernel on the GPU. The mechanism is as follows: The host generates a stream (see Section 12) of instructions which are executed sequentially on the GPU. With

```
cudaEventRecord(event1);
mykernel<<<gridsize, blocksize>>>(...);
cudaEventRecord(event2);
```

three instructions are set on a stream: event 1, the kernel, and event 2. Setting instructions on a stream does not block the host. If the GPU finds an event on the stream, it will store the current GPU time in that event. With

```
cudaEventSynchronize(event2);
```

the host is blocked until the GPU has processed event 2.

You can write an arbitrary number of kernels and events on a stream. They are always processed sequentially. CUDA events are also used for synchronization when multiple streams are used, see Section **??**.

6.2 Cycle Counter on GPU

Each SM has a cycle counter whose value can be queried in a kernel with `clock64()`. The return type is `unsigned long`. Its frequency is the same as the GPU clock frequency. Both CUDA events and `clock64()` measure wall-clock time from the perspective of the GPU. This means that time periods during which a suspended thread is waiting are also counted.

6.3 Wall-Clock on Host

Wall-clock time intervals can be measured with microsecond precision on the host as follows:

```
#include <sys/time.h>

struct timeval start, end;

gettimeofday(&start, NULL);
...
gettimeofday(&end, NULL);

double seconds = (double)(end.tv_sec  - start.tv_sec);
seconds += (double)(end.tv_usec - start.tv_usec)/(1.0e6);
printf("Total elapsed wall-clock time: %f seconds\n", seconds);
```

6.4 Variable Clock Frequency

The clock frequency of a GPU is very low to reduce power consumption when the GPU is idle. Only when the GPU is active, the frequency is increased to its maximum, which takes between 10 and 200ms. This delay has a significant impact on the measured computing time especially for programs with small execution time. One simple solution is to start timing only after the GPU has already been running for some time and reached its maximum frequency. See Section 5 on how to find out the current clock frequency of the SMs.

With some GPUs it is possible to lock the clock frequency to a certain value or range.

```
sudo nvidia-smi -lgc 1200,1800
```

Unlocking is done with

```
sudo nvidia-smi -rgc
```

The same can be done with memory clock frequency:

```
sudo nvidia-smi -lmc 2125,2125
sudo nvidia-smi -rmc
```

On Linux these operations require root privileges.

6.5 Thermal Throttling

If a long series of experiments is carried out, the GPU may heat up and in order to avoid damage, clock frequency is reduced when a certain limit is reached (thermal throttling). Obviously this can lead to wrong time measurements. See Section 5 on how to find out the current GPU temperature and the temperature where throttling sets in.

Some Nvidia drivers log performance reductions and their reasons since the last time the driver was loaded. Run

```
nvidia-smi -q -d performance
```

and check for `Performance State` and `Clocks Throttle Reasons`.

7 Optimization

7.1 Compiler Flags

Compilers are able to optimize code in various ways. The corresponding flag for the GNU C++ compiler is `-O` followed by the optimization level. For example `-O3` gives very good results.

If small inaccuracies with floating point arithmetic are tolearble, the flag `-fast-math` is worth trying. It allows the compiler to reorder floating point operations and ignore exceptional floating point values NaN, Inf and denormalized numbers.

Some flags have to bypass `nvcc` to the underlying GNU compiler. In this case use `-Xcompiler` as for example in

```
nvcc -Xcompiler -mfma program.cu
```

7.2 Loop Unrolling

By default `nvcc` uses loop unrolling for optimization, which has a huge effect in many cases. In order to switch it off for test purposes, write

```
#pragma unroll 1
```

directly before a loop. The number behind `unroll` is the number of times the code in the body of the loop is duplicated.

If a variable, which determines the number of iterations of a loop, is known at compiletime, loops can be unrolled easily. Execution time of programs can therefore be reduced significantly if such variables are replaced by constants whenever possible.

7.3 Restricted Pointers

When a pointer variable is declared with modifier `__restrict__`, the compiler is informed that there is no other way to access the memory region addressed by the pointer. This allows the compiler to make significant optimizations. Without `__restrict__`, the compiler has to assume that two different pointers might be pointing to the same memory location (this is called “aliasing”), which prevents several optimizations.

For example, in a kernel that adds two arrays, `a` and `b` and stores the result in a `c`, the pointers should be declared `__restrict__` to ensure that the compiler can optimize the memory access.

```
__global__ void add
(
  float* __restrict__ a,
  float* __restrict__ b,
  float* __restrict__ c,
  int n
)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
  {
      c[i] = a[i] + b[i];
  }
}
```

7.4 Fused Multiply Add

`nvcc` usually translates a muliply-add operation like

```
d = a*b+c
```

into a single machine instruction. If you want to make it explicit, write

```
d = fmaf(a,b,c)
```

7.5 PTX Machine Code

Sometimes it hard to explain why minor modifications in a program lead to significant changes in computing time. In such cases it can help to look into the PTX code generated by the `nvcc` compiler. PTX stands for Parallel Thread Execution and is a portable, intermediate machine code in ASCII format.

The graphics driver (more concretely `ptxas`, Nvidia's assembler) translates PTX to executable binary code (often called SASS for "streaming assembler") for the specific microarchitecture of the given GPU.

You can generate PTX code from your CUDA program with

```
nvcc -ptx -lineinfo program.cu
```

Flag `-lineinfo` adds references to the lines in the source code, which helps to match PTX instruction with the corresponding CUDA instructions. For example

```
.loc    1 54 7
```

is a reference to line 54 and column 7 in the fist file of the source code.

PTX uses an arbitrary amount of registers, which are declared at the beginning. For example

```
.reg .f32       %f<388>;
```

declares `%f0` ... `%f388` as registers for floating point numbers. The actual mapping to physical registers is done by the graphics driver.

Memory load and store operations (`ld`, verb?st?) can easily be found in a PTX file. For example

```
ld.global.nc.f32    %f17,   [%rd10+8];
```

means loading from global memory register `f17`. The `nc` stands for "non coherent" and means that the value should not be cached. The address is taken from a pointer stored in register `%rd10` plus offset 8 Bytes and the brackets mean dereferencing.

The line

```
fma.rn.f32     %f24, %f23, %f22, %f21;
```

is a floating point (`f32`) fused multiply add (`fma`) operation with round to nearest (`rn`). The order of the registers has to be interpreted as

```
%f24 = %f23 * %f22 + %f21
```

8 Constant Memory

In many cases data is transferred from the host to the GPU and is used read only there. Such data should be written to constant memory on the GPU because constant memory is faster than global memory.

In the following example an array with 1024 floating point numbers shall be copied to constant memory. The array is declared with modifier `__constant__` outside all functions such that it can be used in the main function and the kernel. The host writes data to this array with function `cudaMemcpyToSymbol()`. This has to be done before the grid is launched.

```
// Constant array (pointer to constant device memory)
// Must be declared global. Host writes, kernel reads.
__constant__ float arrayDeviceConst[1024];

__global__ void mykernel()
{
  int tid = threadIdx.x;

  float val = arrayDeviceConst[tid];
}

int main()
{
  // Array on host.
  float arrayHost[1024];

  // Fill array with some data
  for (int i = 0; i < 1024; i++)
    arrayHost[i] = (float)i;

  // Copy arrayHost from host to arrayDeviceConst on device.
  cudaMemcpyToSymbol(arrayDeviceConst, arrayHost, sizeof(arrayHost));

  // Launch grid
  mykernel<<<30, 256>>>();

  // ...
}
```

Constant memory is global in the sense that each SM sees the same content. The reason why it is so fast is that each SM has a local cache for it.

In case you wondered: If a kernel takes arguments, they are always passed via constant memory from the host to the device.

9 Shared Memory and Synchronization

If a host allocates memory block with `cudaMalloc` and passes its address as a function argument to a kernel, then *all threads* running the kernel have access to this block. In that sense, global memory is always shared among all threads. The problem is that global memory is very slow.

Local variables in a kernel are not shared because each thread has its own copy. If you want to share a local variable with the *entire grid*, declare it with modifier `__global__`. The variable is then stored in (slow) global memory.

If a local variable is declared with modifier `__shared__`, it is shared among all threads of the *same block*. It is stored physically in the same fast memory as the L1 cache.

As the threads of block always run in the same SM and shared memory is local to each SM, it makes sense that the variable is shared only within a block. Threads in different blocks can not share memory other than global memory.

In the following example two threads run a kernel `mykernel`, which declares a shared variable `x` and initializes it with 17. Thread 0 writes value 42 to `x` and thread 1 reads `x` and obtains 42. Without modifier `__shared__`, thread 1 would read the initial value 17 for `x`.

```
#include <stdio.h>

__global__ void mykernel()
{
  // Declare shared variable with modifier __shared__.
  __shared__ int x;
  int tid = threadIdx.x;

  x = 17;

  // Synchronize all threads of this block.
  __syncthreads();

  // Thread 0 writes shared variable.
  if(tid == 0) x = 42;

  // Synchronize all threads of this block.
  __syncthreads();

  // Thread 1 reads shared variable.
  if(tid == 1) printf("thread 1 reads shared variable: %d\n",x);
}

int main()
{
  // Launch kernel with 1 block and 2 threads.
  mykernel<<<1, 2>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

In order to avoid a race condition, the two threads have to synchronize to make

sure, that `x` was written by thread 0 before it is read by thread 1. The function `__syncthreads()` synchronizes all threads within a block.

9.1 Race Conditions

Without synchronization, the behavior of a program can be undefined. The following example shows a race condition. Two threads use a shared variable `x`. Both threads write their id to `x` and as the two threads run in parallel, it is undefined whether the value of `x` is 0 or 1 afterwards. Next, both threads read `x` and compare its value to their thread id. One thread detects that the value of `x` was overwritten by the other, but it is undefined which one. If `x` would have been declared without `__shared__`, each thread would have its own copy of `x` and none would overwrite the value of the other. The modifier `volatile` is necessary to prevent the compiler from optimizing `x` away.

```
#include <stdio.h>

__global__ void race()
{
  volatile __shared__ int x;

  int tid = threadIdx.x;

  // Threads write shared variable at the same time.
  // The value of x is afterwards undefined (either 0 or 1).
  x = tid;

  if(x != tid) printf("shared variable was modified by other thread!\n");
}

int main()
{
  // Launch kernel with 1 block and 2 threads.
  race<<<1, 2>>>();
  cudaDeviceSynchronize();
}
```

9.2 Dynamic Shared Memory

Shared memory can be allocated dynamically at runtime. This is necesary when the size of an array is not fixed at compile time. The amount of shared memory in bytes is given as the third parameter after grid- and blocksize when the kernel is launched:

```
  mykernel<<<gridsize, blocksize, sharedmemorysize>>>
```

This means that `sharedmemorysize` bytes are allocated in the shared memory of an SM as soon as a block running `mykernel` is launched there. If not enough memory is free, the block can not be started.

The kernel accesses this shared memory by declaring a local variable for an array with modifier `extern __shared__`:

```
  extern __shared__ int sharedmem[];
```

The value of `sharedmem` is the starting address of the shared memory block with size `sharedmemorysize`.

It makes no sense to declare two variables with `extern __shared__` because they would point to the same memory block. If you want more than one shared array, you have to split it up explicitly, e.g.

```
int* blocka = sharedmem;
int* blockb = sharedmem + 10;
```

If you need shared arrays with different types, you have to cast pointers and take alignment into account. This means that 16/32/64 bit types can only be stored under addresses which are a multiple of 2/4/8.

In the following example an array with random single digit integers is generated. The threads first copy it to shared memory and then apply a median filter. The main reason for using shared memory instead of global memory is faster access. Another important difference is that global memory is shared among all threads in the grid, whereas shared memory is shared only among the threads in a block.

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>

__global__ void median(int* arraydevice, int length)
{
  // sharedmem points to a memoryblock shared by all threads of the block.
  extern __shared__ int sharedmem[];

  int nthreads = blockDim.x;
  int tid = threadIdx.x;
  int i,a,b,c,tmp;

  // Copy arraydevice to sharedmem and synchronize.
  for(i=tid; i<length; i+=nthreads) sharedmem[i] = arraydevice[i];
  __syncthreads();

  // Parallel median.
  for(i=tid+1; i<length-1; i+=nthreads)
  {
    a = sharedmem[i-1];
    b = sharedmem[i];
    c = sharedmem[i+1];

    if(a > b) { tmp=a; a=b; b=tmp; }
    if(a > c) c=a;
    if(b > c) b=c;

    arraydevice[i] = b;
  }
}

int main()
{
  int *arrayhost, *arraydevice;
```

```
  int length = 10;
  int nthreads = 3;
  int size = length*sizeof(int);
  int i;

  arrayhost = (int*)malloc(size);
  cudaMalloc(&arraydevice,size);

  srand((unsigned int)clock());
  for(i=0; i<length; i++)
    arrayhost[i] = (int)(10.0*(float)rand() / (float)RAND_MAX);

  printf("array before filter\n");
  for(i=0; i<length; i++) printf("%d ",arrayhost[i]);

  // Launch kernel with size bytes shared memory in each block.
  cudaMemcpy(arraydevice, arrayhost, size, cudaMemcpyHostToDevice);
  median<<<1, nthreads, size>>>(arraydevice,length);
  cudaDeviceSynchronize();
  cudaMemcpy(arrayhost, arraydevice, size, cudaMemcpyDeviceToHost);

  printf("\narray after filter\n");
  for(int i=0; i<length; i++) printf("%d ",arrayhost[i]);
  printf("\n");

  return 0;
}
```

As mentioned before, shared memory and L1 cache use a common physical memory resource. As the size of the shared memory is not fixed at compile time, it has to be specified which part of this memory shall be used as shared memory. The function

```
cudaFuncSetAttribute
(
  mykernel,
  cudaFuncAttributePreferredSharedMemoryCarveout,
  50*1024
);
```

specifies that 50kB of shared memory/L1 cache should be dedicated for shared memory when running `mykernel`.

The amount of dynamic shared memory a kernel may use per block is limited (usually 48kB by default). This limit can be raised with

```
cudaFuncSetAttribute
(
  mykernel,
  cudaFuncAttributeMaxDynamicSharedMemorySize,
  50*1024
);
```

Now `mykernel` may use up to 50kB dynamic shared memory per block.

9.3 Memory Banks

Shared memory is organized in 32 memory banks. This means that successive word addresses point to successive banks modulo 32. In many cases the 32 threads of a warp access successive words such that each thread addresses a different memory bank. This is a very desirable situation because memory accesses are very fast. A bank conflict arises when two threads in a warp access the same bank. In this case the memory accesses have to be serialized which causes delay. An exception is when several threads of a warp read the *same address* in the same bank. In this case the value is read only once and broadcast without delay.

10 Registers

Registers are the fastest kind of memory. Local variables of a kernel are usually stored there. If there are not enough physical registers available, the variables are spilled out to global memory. This should be avoided because global memory is very slow.

A typical SM has 64K 32 bit registers in total. A single thread may use up to 255 registers. If a thread uses many registers, the number of threads which can run simultaneously on an SM is small, which leads to low occupancy (active blocks per SM). So there is a trade off between register usage and occupancy.

The number of registers which are assigned to a thread and register spilling is determined by the compiler. Register usage and spilling is reported for each kernel with

```
nvcc -Xptxas -v myprogram.cu
```

The runtime of programs can be improved significantly, if registers are used as cache to reduce memory accesses. You can guide the compiler to use many registers per thread and accept low occupancy with

```
__launch_bounds__(ThreadsPerBlock, BlocksPerSM)
```

The value of `ThreadsPerBlock` is the blocksize you are using. If `BlocksPerSM` is 1, you tell the compiler that it's fine if only one block per SM is active and thereby allowing him to assign more registers to a thread.

11 Case Study Convolution

11.1 Mathematical Background

The discrete convolution h of a signal f with an impulse response g is defined by

$$h_\ell = \sum_{m=-\infty}^{\infty} f_{\ell-m} g_m.$$

Assume f and g have a finite length F and G with G much smaller than F. Then

$$\begin{aligned} f_\ell &= 0 \quad \text{for } \ell \notin [0, F-1] \\ g_\ell &= 0 \quad \text{for } \ell \notin [0, G-1] \end{aligned}$$

and

$$h_\ell = \sum_{m=0}^{G-1} f_{\ell-m} g_m.$$

The length of h is $F + G - 1$ meaning

$$h_\ell = 0 \quad \text{for } \ell \notin [0, F+G-2].$$

If f and g are stored in arrays and $f_{\ell-m} g_m$ is accumulated for $m = 0, \ldots, G-1$ in a loop, a case distinction is needed when accessing $f_{\ell-m}$ because

- $\ell - m < 0$ if ℓ is small am m is big
- $\ell - m > F - 1$ if ℓ is big and m is small.

Case distinctions have a very negative performance impact on SIMD hardware. Therefore it is advantageous to extend f by inserting $G-1$ zeros at the beginning and at the end.

If we assume that this has been done in advance, we merely have to compute the values

$$h_\ell = \sum_{m=0}^{G-1} f_{G-1+\ell-m} g_m, \quad \text{for } \ell = 0, \ldots, F-G$$

and no case distinction is necessary in the summation loop.

In order to compute h efficiently on a GPU, the following straight forward approach was taken. Let n be the total number of threads in the system, which is the product of the gridsize and the blocksize. The threads are enumerated consecutively within each block. The i-th thread computes the values for

$$h_i, h_{i+n}, h_{i+2n}, \ldots.$$

This scheme seems to be reasonable because adjacent threads are grouped in a warp. Therefore threads in a warp access in each iteration the same component of g and adjacent components of f, thus providing good memory locality.

11.2 Hardware

The GPU is a Nvidia Geforce RTX 4060 with 28 SMs. Each SM has 128 cores and 4 warp schedulers. The GPU has 8GB global (off chip) memory, 24MB L2 cache for all SMs together and 100kB L1 cache per SM. Further it has 64kB constant memory for all SMs together and 8kB constant memory cache per SM. In order to obtain reproducible results, the SM clock is locked at 2400MHz and the memory clock at 8250MHz.

11.3 Experiments

In the following experiment we chose $F = 2^{23}$ and $G = 512$ such that

$$F - G + 1 = 8\,388\,097$$

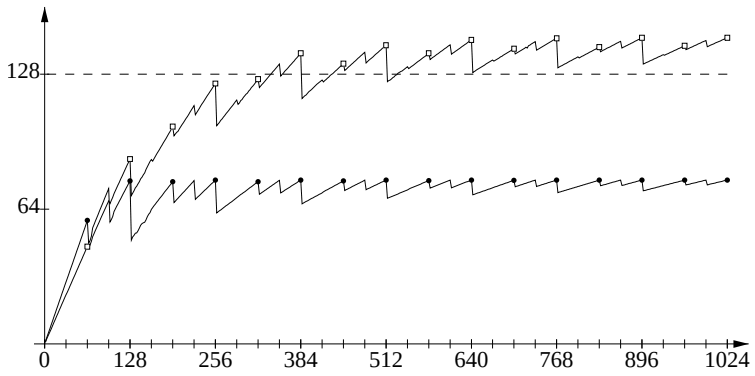
values of h have to be computed, which is is an upper bound for the number of threads we should start. The length of f is 190 seconds at 44.1kHz sampling rate.

Experiments with a single SM

First, we start only a single block with up to 1024 threads, which means that only a single SM is used. In order to obtain a measure for speedup, we divide the computing time with a single thread by the time with n threads.

The time for memory transfer between host and device is not measured and negligible compared to the computing time. As there are 128 cores in an SM one would expect that the speedup will not exceed this limit.

On the horizontal axis in the following graph is the blocksize (number of threads) ranging from 1 to 1024 in logarithmic scale, on the vertical axis is the speedup.



The line with bullets is the straight forward implementation.

- The computig time with a single core was 17 433ms.
- The minimum computing time 224ms was achieved with 1024 threads, which corresponds to a speedup of 77.8.
- Using more than 128 threads gives no significant improvement which is expected as there are only 128 cores.
- The optimal speedup falls short behind 128 which indicates that there must be some bottleneck.
- Further we see that best results are achieved if the number of threads is a multiple of 32 such that there are no empty lanes in warps.

A straight forward improvement is to use fast, constant memory for the pulse response g instead of slow, global memory. The constant cache on each SM is large enough to hold g. The corresponding results are plotted with the boxed line.

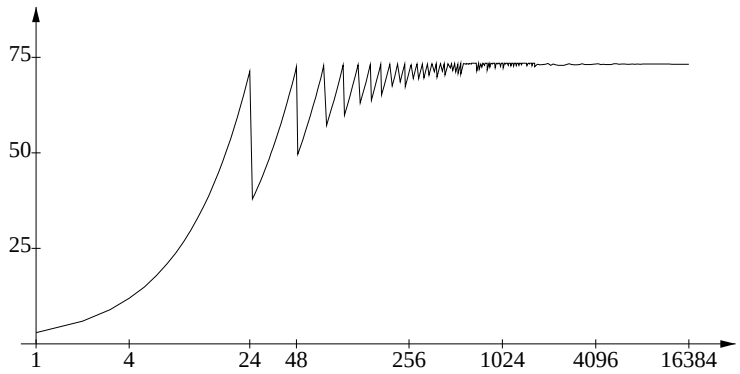
- The computing time with a single core was 22 438ms and is surprisingly longer than without constant memory.

- However, the minimum computing time dropped to only 120ms with 1024 threads. This is a speedup of 187, which is far above the number of 128 cores. The reason is "latency hiding". While one thread has to wait for memory, another thread can use the ALU meanwhile because thread switching costs nothing on GPUs. It worked in our experiment when there were more active threads than cores. Therefore it makes perfectly sense to have many active threads.
- By moving g to constant memory, latency for half of the memory accesses was reduced. As this reduced computing time significantly from 224ms to 120ms, we can conclude that memory latency is the real bottleneck.

Experiments with the entire GPU

According to the literature and our experiments, the number of threads per block should be a multiple of 32 and somewhere between 128 and 512. Provided that sufficiently many blocks are started, computing time does not change significantly as long as the number of threads per block is in this range. We therefore fixed the number of threads per block to 512 and carried out experiments with different numbers of blocks. If the total number of threads (grisize $\times$ blocksize) is equal to the length of h, each thread computes only a single value of h. The number of blocks should therefore be at most $(F - G + 1)/512 \leq 16\,384$.

As a reference for the speedup we used the computing time with one core of an AMD Ryzen 7 7700 with 3GHz clock frequency, which is around 367ms with single precision floating point arithmetic.



Each SM gets the same load if the number of blocks is an integer multiple of the number of SMs. This explains the peaks at $24, 48, 72, \ldots$. The reason why computing time with 25 blocks is significantly higher than with 24 blocks is that a thread block always runs on *the same* SM. This is a key difference to multiprocessor systems, where threads are rescheduled preemptively to any arbitrary, available CPU. Such context switches are expensive because they involve cache coherence protocols and potentially cache pollution.

With the 25 blocks configuration, 23 SMs are each assigned one block, while the final SM is given two. In a worst-case scenario where each block consumes an entire SM's resources, the two blocks on the last SM are executed sequentially. This leads to "tail-end inefficiency", as the other 23 SMs remain idle while the final SM completes its second block.

The load is distributed equally among the SMs as well if the number of blocks is very large. The graph shows that the overhead for a large number of blocks is negligible

and each thread computes only a single value of h. The minimum computing time is $\approx$ 5ms which gives a speedup over the CPU of 73.4 and a speedup over a single core of the GPU of

Summarizing, we obtain the following results:

	Gridsize/Blocksize	Time [ms]	Speedup over CPU
One Core	1/1	22 438	0.016
One SM	1/1024	120	3.06
GPU	16 384/512	5	73.4

One should mention that only one of the eight cores of the CPU was used and time for data transfer between CPU and GPU (5.1ms for f, g and h in total) was neglected. It is also worth noting that such high speedups can only be achieved with single precision floating point arithmetic. Double precision is slower by factor 32 on the GPU, but only by factor 2 on the CPU.

11.4 Optimization

In the simplest version each thread computes only a single element of h. The relevant part of the code for the computation of h_i looks as follows:

```
sum = 0.0;
k = i+ng-1;
for(j=0; j<ng; j++)
{
  sum += f[k] * g[j];
  --k;
}
h[i] = sum;
```

In each iteration of the j-loop, two memory accesses are needed for one multiply-add operation. Therefore most of the computing time is wasted waiting for memory.

A better approach is to assign the computation of x consecutive values of h to each thread. So, the i-th thread now computes values

$$h_i, h_{i+1}, h_{i+2}, \ldots, h_{i+x-1}.$$

for $i = 0, x, 2x, 3x, \ldots$. The important point is that the values are not computed sequentially but interleaved:

```
for(m=0; m<x; m++) sum[m] = 0.0;

k = i+ng-1;
for(j=0; j<ng; j++)
{
  for(m=0; m<x; m++) sum[m] += f[k+m]*g[j];
  --k;
}

for(m=0; m<x; m++) h[i+m] = sum[m];
```

At a first glance, there seems to be no benefit. However, after the thread has loaded a value of f from memory, it can keep it in a register and use it for the computation of x values of h! This means only two memory accesses for x multiply-add operations.

However, there is a problem: Keeping this "sliding window" of f in registers requires that the registers are shifted in each iteration. A new value of f is loaded, and an old value of f can be dropped. This costs $x - 1$ register to register copy operations in each iteration. Now, let us unroll the entire j-loop, meaning that separate code is generated for each iteration using `#pragma unroll`. This means that the compiler produces code which takes the "right" registers in each iteration and avoids the register copy operations entirely. The code will be the same in every x-th iteration of the j-loop. Of course a diligent programmer can do this as well (or better write a programm which produces that code).

This modification reduces execution time significantly from **5ms** to only **0.83ms** and shows, that confirms that so far most of the time was lost waiting for memory.

The total number of multiply-add operations for the convolution with $F = 2^{23}$ and $G = 512$ is roughly

$$2^{23} \times 512 \quad = \quad 2^{32}.$$

This makes

$$\frac{2^{32} \times 2 \text{ FLOP}}{0.83\text{ms}} \quad = \quad 10.35 \text{ TFLOP}/s$$

The theoretical peak floating point performance of the RTX 4060 at 2.4GHz clock frequency with 24 SMs and 128 cores per SM is

$$\frac{2.4 \times 10^9}{s} \times 24 \times 128 \times 2 \text{ FLOP} \quad = \quad 14.7 \text{ TFLOP}/s$$

provided that one addition and one multiplication are executed in each cycle. This means that the optimized version comes rather close to the theoretical limit.

The speedup over the CPU is now 442 provided the time for memory transfer between CPU and GPU is not taken into account.

11.5 Further Considerations on Efficiency

A well known reason for delays are "bubbles" in the arithmetic and instruction pipeline. For example, there is a data dependency in the summation variable when computing a dot product. Fortunately the ALUs in modern GPUs use dedicated hardware for "data forwarding", such that dot products can be computed without stalling the pipeline.

Branch operations, which occur in each iteration of a loop, are a problem for instruction pipelines. An optimizing compiler like `nvcc` applies all kinds of tricks like loop unrolling and branch prediction to cope with this. Whenever possible one should make sure that the length of inner loops is a compile time constant because this makes loop unrolling easier.

12 Streams

A CUDA stream is a sequence of GPU operations (kernel launches, memory transfers, events) that are executed sequentially in the order they are issued by the host. By default, all operations are put into Stream 0. When the host writes an operation to a stream, it is not blocked. As an example see Section 6.1 where CUDA events were added to the default stream.

The real power of streams comes when you use multiple streams. When you issue operations to different streams, they can be executed concurrently as long as there are no data dependencies between them. For example, you can run a kernel on one part of the GPU while simultaneously transferring data from the host to the device in another part, provided that these two operations don't interfer with each other. This is crucial for overlapping computation and data transfer, which can significantly improve overall performance, see Section 12.1

You can execute kernels sequentially by writing them to the same stream or in parallel by using a different stream for each kernel.

Streams are independent. To ensure an operation in one stream doesn't start before an operation in another stream finishes, you must explicitly synchronize them. This is done using functions like `cudaStreamSynchronize()` or `cudaEventRecord()` and `cudaEventSynchronize()`.

In the following example two streams are used such that kernel 3 runs parallel to kernel 1 and 2, which are executed sequentially.

```
#include <stdio.h>

__global__ void kernel1(){ }
__global__ void kernel2(){ }
__global__ void kernel3(){ }

int main()
{
  cudaStream_t streamA, streamB;
  cudaStreamCreate(&streamA);
  cudaStreamCreate(&streamB);

  // Launch kernel1 and kernel2 in streamA.
  kernel1<<<32, 256, 0, streamA>>>();
  kernel2<<<32, 256, 0, streamA>>>();

  // Launch kernel3 in streamB.
  kernel3<<<32, 256, 0, streamB>>>();

  // Wait for both streams to finish
  cudaStreamSynchronize(streamA);
  cudaStreamSynchronize(streamB);

  cudaStreamDestroy(streamA);
  cudaStreamDestroy(streamB);

  return 0;
}
```

12.1 Asynchronous Memory Transfer

Memory transfers can also be added to a stream. If you want to run several streams in parallel, you have to use *asynchronous* memory transfer. This means that the host initiates the transfer but does not wait for its completion. This is achieved with funciton `cudaMemcypAsync`. In order to avoid that the host operating system swaps out the memory pages before the transfer is complete, you have to allocate memory on the host with `cudaMallocHost`.

```
int main()
{
  cudaStream_t stream;
  cudaStreamCreate(&stream);
  float * arrayhost, *arraydevice;
  int length = 1024;
  int size = length*sizeof(float)M

  cudaMallocHost(&arrayhost,size); // Memory is not swapped out by host OS.
  cudaMalloc(&arraydevice,size);

  // Add memory transfers and kernel to stream without blocking host.
  cudaMemcpyAsync(arraydevice,arrayhost,size,cudaMemcypHostToDevice,stream);
  kernel<<<32,256,0,stream>>>(arraydevice,length);
  cudaMemcpyAsync(arrayhost,arraydevice,size,cudaMemcypDeviceToHost,stream);

  // Host waits for stream to finish.
  cudaStreamSynchronize(stream);
}
```

13 Tensor Cores for Matrix Multiplication

Modern GPUs have several tensor cores per SM. A tensor core is a hardware matrix-multiply unit. It operates per warp on fragments of a matrix called tiles. These are usually 16×16 submatrices.

13.1 Matrix Tiles

The straight forward way to multiply two matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$ in CUDA is that each thread computes one component of the product $C \in \mathbb{R}^{m \times n}$ according to the "row by column" rule:

$$c_{ij} \quad = \quad \sum_{\ell=0}^{k-1} a_{i\ell} b_{\ell j}$$

Every multiply-add operation in this loop is a scalar operation, meaning one multiply-add instruction per cycle per thread.

The problem with this approach is that there are way too many memory accesses because the components of A and B are loaded repeatedly by different threads. For each component c_{ij} of C there are $2k$ memory accesses to load $a_{i\ell}$ and $b_{\ell j}$ for $\ell = 0, \ldots, k-1$. As C has mn components, the total amount of memory accesses including storage of C is

$$mn(1 + 2k).$$

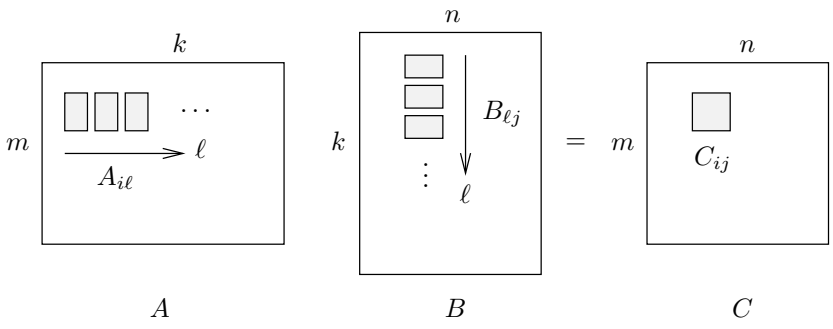
A better way is to split A, B, C into smaller tiles, e.g. 16×16 submatrices. If m, n, k are not multiples of 16, the matrices have to be zero padded. The i, j-th tile of A has components

$$A_{ij} \quad = \quad \begin{pmatrix} a_{16i,16j} & \cdots & a_{16i,16j+15} \\ \vdots & \ddots & \vdots \\ a_{16i+15,16j} & \cdots & a_{16i+15,16j+15} \end{pmatrix}$$

and correspondingly for B and C. The tile C_{ij} is now obtained by

$$C_{ij} \quad = \quad \sum_{\ell=0}^{(k-1)/16} A_{i\ell} B_{\ell j}$$

where the product $A_{i\ell} B_{\ell j}$ is a 16×16 matrix multiplication.



Assuming that a tile is loaded only once during this multiplication we need 2×16^2 memory operations to load $A_{i\ell}$ and $B_{\ell j}$. The computation of C_{ij} amounts to

$$2 \times 16^2 \times \frac{k}{16} \quad = \quad 32k$$

memory operations. As C consists of $mn/16^2$ tiles, the total amount including storage of C is

$$32k\frac{mn}{16^2} + mn \quad = \quad mn(1 + k/8)$$

memory accesses, which is roughly **factor 16** less than with the straight forward approach, assuming k is large.

The crucial assumption was that the tiles $A_{i\ell}$ and $B_{\ell j}$ are loaded only once for the computation of the tile product $A_{i\ell}B_{\ell j}$. There are two options:

- Store the tiles in shared memory. This costs repeated memory accesses but at least they are faster.
- Store the tiles in registers of a thread. This means no parallelism during computation of a tile product.

Tensor cores combine the advantages of both options: They store the tiles in registers but use an entire warp of threads to compute the tile product in parallel.

What is the optimal tile size given that fast memory is limited? In the general case we may use $m' \times n'$ tiles of C and correspondingly $m' \times k'$ tiles of A and $k' \times n'$ tiles of C. Loading a pair of tiles from A and B takes

$$m' \times k' + k' \times n' \quad = \quad k'(m' + n')$$

memory accesses. The computation of a tile of C costs

$$k'(m' + n') \times \frac{k}{k'} \quad = \quad k(m' + n')$$

accesses. As C consists of $mn/(m'n')$ tiles, the total amount including storage of C is

$$k(m' + n')\frac{mn}{m'n'} + mn \quad = \quad mn\left(1 + k\frac{m' + n'}{m'n'}\right)$$

memory accesses. The result is independent of k', which means that $k' = 1$ is the best choice to reduce the amount of fast memory needed to store tiles of $A_{i\ell}$ and $B_{\ell j}$. In this case $A_{i\ell}B_{\ell j}$ is a dyadic product of a column vector and a row vector. For a given tile area $m'n'$, which corresponds to the required amount of fast memory for a tile of C, a squared shape with $m' = n'$ minimizes the number of memory operations.

13.2 Tensor Cores

A tensor core is a special functional unit inside an SM for multiplying and accumulating small (e.g. 16×16) matrix tiles. It is efficient because the tiles are stored in registers and the product is computed with a high degree of parallelism with a single PTX instruction.

Computation time is reduced even more at the cost of accuracy. A floating point number uses a certain number of bits for the mantissa. The exact product of two such numbers would require twice the number of bits such that after rounding to the original

format, half of them are lost. This is a waste which suggests that not much accuracy is lost if only the most significant half of mantissa bits are actually multiplied and no rounding is necessary. Tensor cores provide therefore less accurate floating point formats for the argument matrices A and B:

- half precision FP16: 10 bit mantissa, 5 bit exponent,
- tensor float TF32: 10 bit mantissa, 8 bit exponent or
- brain float BF16: 7 bit mantissa, 8 bit exponent,

The output is accumulated in

- single precision FP32: 23 bit mantissa, 8 bit exponent.

In all cases one bit is used for the sign.

A matrix multiply add (MMA) operation with a tensor core happens per warp: All 32 threads collaborate to load a tile of A and B into the registers of a tensor core. If the warp issues the MMA instruction, the hardware computes

$$D \quad = \quad AB + C$$

for an entire tile, which means order of 16^3 FMA instructions in just a few cycles.

There is a C++ API for loading tiles from memory into registers and executing the matrix multiply add operation on a tensor core called WMMA (Warp Matrix Multiply and Accumulate) [9], which is described in Section 13.3. If you just want to solve linear algebra problems using GPU's and are not interested in how tensor cores work, it is easier to use high level libraries like cuBLAS and cuBLASLt, which are described in Section 14 and 15.

13.3 Warp Matrix Multiply and Accumulate (WMMA)

WMMA is a C++ API enclosed in a namespace which can be used with

```
using namespace nvcuda::wmma;
```

If you want to multiply two matrix tiles $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$ and accumulate the result in $C \in \mathbb{R}^{m \times n}$, you first have to declare the tile sizes and the types of the input matrices A and B and the accumulator. This is done with

```
fragment<matrix_a, Mtile, Ntile, Ktile, half, row_major> a_frag;
fragment<matrix_b, Mtile, Ntile, Ktile, half, col_major> b_frag;
fragment<accumulator, Mtile, Ntile, Ktile, float> c_frag;
```

The values of `Mtile`, `Ntile` and `Ktile` are the matrix sizes m, n, k. Only certain combinations for (m, n, k) are supported like e.g. $(16, 16, 16)$, $(32, 8, 16)$ or $(8, 32, 16)$. Note that these values are not passed as arguments to a constructor of `a_frag` or `b_frag` but as parameters of a C++ template. This means that in general `a_frag`, `b_frag` and `c_frag` have different types. `half` and `float` are the types of A, B and C. The tensor core is instructed to store the fragments in its internal registers in row major format for A and column major format for B. According to Nvidia this gives best performance.

The accumulator matrix is initialized with zero, which is done with

```
fill_fragment(c_frag, 0.0f);
```

Next, A and B are loaded with

```
load_matrix_sync(a_frag, a, Astride, mem_row_major);
load_matrix_sync(b_frag, b, Bstride, mem_row_major);
```

Here `a` and `b` are the addresses of a fragment of the matrices and have type `half*`. Usually the tiles are fragments of larger matrices. In this example both matrices are stored row major, which means that elements of a row are stored consecutively in an array. This means that when loading B the tile will be transposed. It is therefore recommended to store B column major whenever possible. The value of `Astride` and `Bstride` is the number of rows of the large matrices, of which A and B are fragments. These values are needed such that the system can calculate the start addresses of the rows of the fragments. (If column major format is used, the strides are the number of columns.)

The actual matrix multiply and accumulate operation is done with

```
mma_sync(c_frag, a_frag, b_frag, c_frag);
```

Note that the added matrix is the same as the accumulator in this example.

Finally, the accumulator is written to memory with

```
store_matrix_sync(c, c_frag, Cstride, mem_row_major);
```

Again we have to provide the format (in this case row major), a pointer `c` to the position where C is stored of type `float*` and the stride of the larger matrix of which C is a fragment.

14 cuBLAS Library for Linear Algebra

The cuBLAS library [1] is an implementation of linear algebra functions like matrix multiplication using CUDA. It gives high level access to the computational resources of Nvidia GPUs including Tensor Cores.

14.1 Example: Matrix Multiplication with cuBLAS

A simple example for computing

$$C \leftarrow \alpha AB + \beta C$$

where $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$ and $C \in \mathbb{R}^{m \times n}$ is as follows:

```
#include <stdio.h>
#include <cublas_v2.h>
#include <cuda_fp16.h>

#define M 16
#define N 16
#define K 16

int main()
{
  // Host memory
  float Ahost[M*K], Bhost[K*N], Chost[M*N];
  float alpha = 1.0f, beta = 0.0f;

  // Initialize input matrices with ones.
  for (int i=0; i<M*K; i++) Ahost[i] = 1.0f;
  for (int i=0; i<K*N; i++) Bhost[i] = 1.0f;
  for (int i=0; i<M*N; i++) Chost[i] = 1.0f;

  // Device memory
  float *Adevice, *Bdevice, *Cdevice;
  cudaMalloc(&Adevice, M*K*sizeof(float)); // M rows, K cols
  cudaMalloc(&Bdevice, K*N*sizeof(float)); // K rows, N cols
  cudaMalloc(&Cdevice, M*N*sizeof(float)); // M rows, N cols

  // Copy matrices to device.
  cudaMemcpy(Adevice, Ahost, M*K*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(Bdevice, Bhost, K*N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(Cdevice, Chost, M*N*sizeof(float), cudaMemcpyHostToDevice);

  // cuBLAS handle
  cublasHandle_t handle;
  cublasCreate(&handle);

  // Enable Tensor Core acceleration (TF32 in this case)
  cublasSetMathMode(handle, CUBLAS_TF32_TENSOR_OP_MATH);

  // Compute C = alpha * A * B + beta * C
  cublasSgemm
```

```
  (
    handle, CUBLAS_OP_N, CUBLAS_OP_N,
    M, N, K,
    &alpha,
    Adevice, M,    // A is MxK, M rows.
    Bdevice, K,    // B is KxN, K rows.
    &beta,
    Cdevice, M     // C is MxN, M rows.
  );
  cudaDeviceSynchronize();


  // Copy result to host.
  cudaMemcpy(Chost, Cdevice, M*N*sizeof(float), cudaMemcpyDeviceToHost);

  printf("cuBLAS result C[0,0] = %f\n",Chost[0]);

  // Cleanup
  cublasDestroy(handle);
  cudaFree(Adevice); cudaFree(Bdevice); cudaFree(Cdevice);

  return 0;
}
```

The cuBLAS library has to be linked with

```
  nvcc matrix.cu -lcublas
  nvcc matrix.cu cublas.lib
```

The second version is for Windows. The cublas library under windows is in

```
  C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\vXX.X\lib\x64
```

The meaning of the instructions is as follows: First, a handle to cuBLAS has to be created with

```
  cublasHandle_t handle;
  cublasCreate(&handle);
```

With function

```
  cublasSetMathMode(handle, mode);
```

you can select the floating point precision. The most important modes are as

- `CUBLAS_DEFAULT_MATH`. Highest performance, uses Tensor Cores.
- `CUBLAS_PEDANTIC_MATH`. Uses strict IEEE754 precision for the given floating point types. Tensor Cores are not used because they compute internally with lower precision.
- `CUBLAS_TF32_TENSOR_OP_MATH`. Enables acceleration of single-precision routines using TF32 floating point format in Tensor Cores.

The actual matrix multiplication is done with

```
cublasStatus_t cublasSgemm
(
  cublasHandle_t handle,
  cublasOperation_t transA, cublasOperation_t transB,
  int m, int n, int k,
  const float *alpha,
  const float *A, int nrowsA,
  const float *B, int nrowsB,
  const float *beta,
  float *C, int nrowsC
)
```

With paramenters `transA` and `transB` you can transpose A and B prior to the multiplication. Value `CUBLAS_OP_N` means do not transpose, value `CUBLAS_OP_T` means transpose. The parameters can also be used if one or both matrices have been stored row-major and need to be reorganized as column-major without actually doing the transposition.

A matrix is always stored column-major as one-dimensional array in cuBLAS, which means that the components of a column are contiguously in memory. An $m \times n$ matrix

$$C = \begin{pmatrix} c_{11} & \dots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{m1} & \dots & c_{mn} \end{pmatrix}$$

is storead as

$$[c_{11}, c_{21}, \dots, c_{m1},\ c_{12}, c_{22}, \dots, c_{m2},\ \dots\ c_{1n}, c_{2n}, \dots, c_{mn}]\,.$$

The "leading dimension" of a matrix stored column-major is the number of its rows (its height) or the length of a column. It can also be defined as the stride between successive columns in memory.

The parameters `nrowsA`, `nrowsB`, `nrowsC` are the leading dimensions of the arrays `A`, `B` and `C`. In the above example we have $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$ and $C \in \mathbb{R}^{m \times n}$ and therefore the leading dimensions are m, k, m respectively.

Every cuBLAS function returns an error code of type

```
cublasStatus_t
```

If `CUBLAS_STATUS_SUCCESS` is returned, the function was executed with no error. The meaning of other error codes is described in [1].

14.2 Parallel Streams with cuBLAS

If you want to execute two cuBLAS functions in parallel, you can use streams in the same way as with CUDA: Generate two streams and two cuBLAS handles, which are associated with the streams as follows:

```
cudaStream_t stream1, stream2;
cublasHandle_t handle1, handle2;
```

```
// Generate two cuBLAS handles and two streams.
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cublasCreate(&handle1);
cublasCreate(&handle2);

// Associate handle1 with stream1 and handle2 with stream 2.
cublasSetStream(handle1, stream1);
cublasSetStream(handle2, stream2);

// Execute cuBLAS functions with different handles.
...

// Wait for termination.
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

When a cuBLAS function is called with handle1 or handle2, it is placed on the corresponding stream1 or stream2. Functions in different streams are executed in parallel, functions in the same stream sequentially.

15 cuBLASLt Library for Linear Algebra

The cuBLASLt (cuBLAS lightweight) library is the successor of cuBLAS. It supports more floating point types and is optimized to use Tensor Cores efficiently. While cuBLAS is stable and well established, cuBLASLt is the recommended path forward.

15.1 Example: Matrix Multiplication with cuBLASLt

A simple example for computing

$$D \leftarrow \alpha AB + \beta C$$

where $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$ and $C \in \mathbb{R}^{m \times n}$ is as follows:

```
#include <cublasLt.h>
#include <stdio.h>

#define N 17
#define M 26
#define K 19

int main()
{
  // Host memory
  float Ahost[M*K], Bhost[K*N], Chost[M*N], Dhost[M*N];
  float alpha = 1.0f, beta = 0.0f;

  // Initialize input matrices with ones.
  for (int i=0; i<M*K; i++) Ahost[i] = 1.0f;
  for (int i=0; i<K*N; i++) Bhost[i] = 1.0f;
  for (int i=0; i<M*N; i++) Chost[i] = 1.0f;

  // Device memory
  float *Adevice, *Bdevice, *Cdevice, *Ddevice;
  cudaMalloc(&Adevice, M*K*sizeof(float)); // M rows, K cols
  cudaMalloc(&Bdevice, K*N*sizeof(float)); // K rows, N cols
  cudaMalloc(&Cdevice, M*N*sizeof(float)); // M rows, N cols
  cudaMalloc(&Ddevice, M*N*sizeof(float)); // M rows, N cols

  // Copy matrices to device.
  cudaMemcpy(Adevice, Ahost, M*K*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(Bdevice, Bhost, K*N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(Cdevice, Chost, M*N*sizeof(float), cudaMemcpyHostToDevice);

  // Create cuBLASLt handle.
  cublasLtHandle_t ltHandle;
  cublasLtCreate(&ltHandle);

  // Operation descriptor: FP32 compute, FP32 scale
  cublasLtMatmulDesc_t operationDesc;
  cublasLtMatmulDescCreate
  (
    &operationDesc,
```

```
    CUBLAS_COMPUTE_32F_FAST_TF32, // Precision for intermediate results.
    CUDA_R_32F                    // Precision for alpha and beta.
  );

  // Matrix layouts, column-major.
  cublasLtMatrixLayout_t layoutA, layoutB, layoutC, layoutD;
  cublasLtMatrixLayoutCreate(&layoutA, CUDA_R_32F, M, K, M);
  cublasLtMatrixLayoutCreate(&layoutB, CUDA_R_32F, K, N, K);
  cublasLtMatrixLayoutCreate(&layoutC, CUDA_R_32F, M, N, M);
  cublasLtMatrixLayoutCreate(&layoutD, CUDA_R_32F, M, N, M);

  // Compute D = alpha*A * B + beta*C
  cublasLtMatmul
  (
    ltHandle,
    operationDesc,     // Operation descriptor.
    &alpha,
    Adevice, layoutA,  // Matrix with its layout.
    Bdevice, layoutB,
    &beta,
    Cdevice, layoutC,
    Ddevice, layoutD,
    NULL, nullptr, 0,  // Optimized algorithm.
    0                  // Stream.
  );
  cudaDeviceSynchronize();

  // Copy result.
  cudaMemcpy(Dhost, Ddevice, M*N*sizeof(float), cudaMemcpyDeviceToHost);
  printf("cuBLASLt result D[0,0] = %f\n",Dhost[0]);

  // Cleanup
  cublasLtMatmulDescDestroy(operationDesc);
  cublasLtMatrixLayoutDestroy(layoutA);
  cublasLtMatrixLayoutDestroy(layoutB);
  cublasLtMatrixLayoutDestroy(layoutC);
  cublasLtMatrixLayoutDestroy(layoutD);
  cublasLtDestroy(ltHandle);

  cudaFree(Adevice); cudaFree(Bdevice); cudaFree(Cdevice); cudaFree(Ddevice);
  return 0;
}
```

Compile with

```
nvcc matrix.cu -lcublasLt
```

Compared to the matrix multiplication example with cuBLAS in Section 14.1, there are some differences. First, the handle to the library is now created with

```
cublasLtHandle_t ltHandle;
cublasLtCreate(&ltHandle);
```

In cuBLASLt operation descriptors are used to set parameters for an operation, in

this case matrix multiplication. In the following example, the floating point formats for intermediate results and for α and β are set.

```
cublasLtMatmulDesc_t operationDesc;
cublasLtMatmulDescCreate
(
  &operationDesc,
  CUBLAS_COMPUTE_32F_FAST_TF32, // Type for intermediate results.
  CUDA_R_32F                   // Type for alpha and beta.
);
```

We use `CUBLAS_COMPUTE_32F_FAST_TF32` which allows to use tensor cores with single precision (F32) input matrices and tensor float (TF32) format for intermediate results. The type `CUDA_R_32F` means single precision (F32) floating point numbers for the scalars α and β

Next, for each matrix a layout is specified, for example

```
cublasLtMatrixLayout_t layoutA
cublasLtMatrixLayoutCreate(&layoutA, CUDA_R_32F, M, K, M);
```

The layout describes a $m \times k$ matrix with single precision floating point format. It is stored column-major, therefore the leading dimension is equal to the number m of rows, which is passed as the last argument.

The next operation is the actual computation of $D \leftarrow \alpha AB + \beta C$. This function is declared as follows:

```
cublasStatus_t cublasLtMatmul
(
  cublasLtHandle_t           handle,
  cublasLtMatmulDesc_t       operationdescr,
  const void                 *alpha,
  const void                 *A,
  cublasLtMatrixLayout_t     layoutA,
  const void                 *B,
  cublasLtMatrixLayout_t     layoutB,
  const void                 *beta,
  const void                 *C,
  cublasLtMatrixLayout_t     layoutC,
  void                       *D,
  cublasLtMatrixLayout_t     layoutD,
  const cublasLtMatmulAlgo_t *algo,
  void                       *workspace,
  size_t                     workspaceSizeInBytes,
  cudaStream_t               stream
);
```

Note that the matrices are passed as `void` pointers. The reason is that many different types for floating point values are provided by cuBLASLt and the actual type is given by the layout descriptor for each matrix. The same holds for `alpha` and `beta`, whose type is given by the operation descriptor.

If you want to overwrite C with the result, passing the same address for C and D will work and avoids allocating extra memory for D.

The last argument is a CUDA stream. If two cuBLASLt functions are called with different streams, they are executed in parallel. The value 0 as in the example above denotes the default stream. Note that in cuBLASLt the stream is passed as an argument to the matrix multiplication function and is *not* associated with the handle as in cuBLAS. Therefore no function `cublasLtSetStream` exists in cuBLASLt.

With argument `algo` an optimized algorithm for matrix multiplication can be chosen. There are several matrix multiplication algorithms which differ in the following aspects:

- Use Tensor Cores or CUDA threads only.
- How the matrices are decomposed into tiles, which are processed by Tensor Cores.
- Some algorithms need additional memory for intermediate results (workspace).
- Some algorithms are non-deterministic, for example when warp-scheduling influences the order of summation.
- Some algorithms are optimized for matrices whose number of rows and colmns differ greatly, others for near square matrices.

With function `cublasLtMatmulAlgoGetHeuristic()` the best algorithm can be selected depending for example on the matrix shapes and selected floating point types. If, as in the above example, a NULL pointer is passed for `algo`, then a default algorithm based on hardware capability, matrix sizes, and data types is used. In the above example no workspace for intermediate results is provided.

References

[1] *cuBLAS API Documentation*. – https://docs.nvidia.com/cuda/cublas/

[2] *CUDA and Floating Point*. – https://docs.nvidia.com/cuda/floating-point/index.html#cuda-and-floating-point

[3] *Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. – https://docs.nvidia.com/cuda/floating-point/index.html

[4] *Flynn's Taxonomy*. – https://en.wikipedia.org/wiki/Flynn%27s_taxonomy

[5] *GPU Programming with CUDA*. – https://vstahl.4lima.de/cuda/vorlesung.html

[6] *Parallel Thread Execution ISA*. – https://docs.nvidia.com/cuda/parallel-thread-execution/

[7] *IEEE Standard for Binary Floating-Point Arithmetic*. 1985. – https://www.ime.unicamp.br/~biloti/download/ieee_754-1985.pdf

[8] *IEEE Standard for Floating-Point Arithmetic*. 2019. – https://www-users.cse.umn.edu/~vinals/tspot_files/phys4041/2020/IEEE%20Standard%20754-2019.pdf

[9] Gautam, Tushar: *Introduction to Tensor Cores Programming*. – https://0mean1sigma.com/tgemm/

[10] Nathan Whitehead, Alex Fit-Florea: *NVIDIA CUDA Compiler Driver NVCC*. – https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

[11] Nathan Whitehead, Alex Fit-Florea: *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. – https://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf

Index