In Memory of Nacho Navarro

PUMPS+AI 2022

Lecture 6: Advanced Features:
Tensor Cores, WARP programming, Unified Memory

Wen-mei Hwu, Mateo Valero, Toni Pena,
Juan Gomez-Luna, Marc Jorda, Leonidas Kosmidis, Bernat Font
Joint Register and Shared Memory Tiling

- Store input M tile and output P tile elements in registers
- Store input N tile elements in shared memory
- Decouple of M and N input tile widths
  - TILE_WIDTH_M, TILE_WIDTH_N
- Key quantities
  - Number of threads = TILE_WIDTH_M
  - Output tile size = TILE_WIDTH_M * TILE_WIDTH_N
  - Reuses for each N element = TILE_WIDTH_M
  - Reuses for each M element = TILE_WIDTH_N
  - Each thread calculates TILE_WIDTH_N P elements

Example:
TILE_WIDTH_M = 4
TILE_WIDTH_N = 2
Deep Learning Matrix Multiplication
Hierarchical Decomposition

https://www.anandtech.com/show/12673/titan-v-deep-learning-deep-dive/3
Volta is equipped with 640 Tensor Cores
- Each performing 64 FMA operations per clock

Mixed-precision matrix multiply in FP16 and FP32
- 12x higher peak teraflops than Pascal
- 3x performance speedups in training over Pascal
NVIDIA TensorCores

Google's TPU

Apple's A11

Intel's DLBoost

Many many startups

NVIDIA V100 Subcore. Each SM contains 4 subcores and there are 80SMs on the chip. In total there are 640 TCU elements.
Load/Store from TensorCore

Fixed size matrix multiplication: usually 4x4, 16x16, or 64x64

NVIDIA supports non-square matrix dimensions
Fine-grained structured sparsity for Tensor Cores
• 50% fine-grained sparsity
• 2:4 pattern: 2 values out of 4 must be zero

Addresses the three challenges:
• **Accuracy**: maintains accuracy of the original, unpruned network
• Medium sparsity level (50%), fine-grained

• **Recipe**: shown to work across tasks and networks

• **Speedup**
• Math: Specialized Tensor Core support for sparse math
• Structured nature allows aligned, regular memory accesses and minimal metadata overheads
Compressed Matrix:
• Data: ½ size
• Metadata:
  2b per nonzero element (index into its location in the 4-element region)
16b data: 12.5% overhead
8b data: 25% overhead
TensorCore WMMA
(WARP Matrix Multiply Accumulate) API

wmma::fragment<_Use, _M, _N, _K, _Type, _Layout>;

wmma::fill_fragment(fragment<...> &, const _Type &);

wmma::load_matrix_sync(fragment<...> &, const _Type *, unsigned, _Layout);

wmma::store_matrix_sync(_Type *, const fragment<...> &, unsigned, _Layout);

wmma::mma_sync(fragment<...> &, const fragment<...> &, const fragment<...> &, const fragment<...> &, bool);

Fairly simple low level API that operates on memory fragments (internally registers)
A memory fragment (internally a set of registers)
wmma::fill_fragment(
    fragment<...> & fragment,  // Fragment to fill */
    const _Type & C            // Constant value to fill with */
);

Fills the fragment with some constant value C
Loading Matrix Fragment from Memory

```cpp
wmma::load_matrix_sync(
    fragment<...> & fragment,    // Fragment to load */
    const _Type * mptr,           // Staring address to load from */
    unsigned lda,                 // Leading matrix dimension */
    _Layout layout;              // Matrix layout */
);

wmma::load_matrix_sync(a_frag, &a[a_row + a_col * lda], lda);
```

**Loads data from global or shared memory with specified stride into the fragment**

If the fragment is `matrix_a` or `matrix_b`, the layout is inferred from the fragment type.

In the example, we load a tile of a matrix (at `&a[aRow+aCol*lda]`) into `a_frag` with `lda` as the stride.
Stores the fragment into global or shared memory with specified stride and layout
Stores in **out** the result of computing \( a \times b + c \)

If **satf** is true, the following saturation happens:

- Infinite saturates to **MAX_NORM**
- Minus infinite saturates to \(-\text{MAX_NORM}\)
- NaN is transformed to 0
GEMM Kernel Using MMA in Half Precision

(1 of 3)

```cpp
#include <mma.h>

using namespace nvcuda::wmma;

__global__ void dot_wmma(half * a, half * b, half * c,
                        int M, int N, int K, float alpha, float beta) 
{
    int lda = M; int ldb = K; int ldc = M;
    int warp_m = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
    int warp_n = (blockIdx.y * blockDim.y + threadIdx.y);

    fragment<matrix_a, 16, 16, 16, half, col_major> a_frag;
    fragment<matrix_b, 16, 16, 16, half, row_major> b_frag;
    fragment<accumulator, 16, 16, 16, half> acc_frag;
    fragment<accumulator, 16, 16, 16, half> c_frag;

    fill_fragment(acc_frag, 0.0f);
}
```
for (int i = 0; i < K; i += 16) {
    int a_row = warp_m * 16; int a_col = i;
    int b_row = i; int b_col = warp_n * 16;

    if (a_row < M && a_col < K && b_row < K && b_col < N) {
        /* Load fragments in the tile */
        load_matrix_sync(a_frag, a + a_row + a_col * lda, lda);
        load_matrix_sync(b_frag, b + b_row + b_col * ldb, ldb);

        /* Compute partial result and accumulate */
        mma_sync(acc_frag, a_frag, b_frag, acc_frag);
    }
}
int c_row = warp_m * 16; int c_col = warp_n * 16;

if (c_row < M && c_col < N) {
    /* Load output fragment from memory */
    load_matrix_sync(c_frag, c + c_row + c_col * ldc, ldc, mem_col_major);

    /* Update fragment with computed result */
    for (int i = 0; i < c_frag.num_elements; i++) {
        c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
    }

    /* Store output fragment */
    store_matrix_sync(c + c_row + c_col * ldc, c_frag, ldc, mem_col_major);
}
}
RECENT INNOVATIONS IN ON-CHIP MEMORY
Example of data movement between GPU global memory (DRAM) and GPU cores.

NVIDIA A100 Tensor Core GPU Architecture

A100 improves SM bandwidth efficiency with a new load-global-store-shared asynchronous copy instruction that bypasses L1 cache and register file (RF). Additionally, A100’s more efficient Tensor Cores reduce shared memory (SMEM) loads.

A100 feature:
Direct copy from L2 to scratchpad, bypassing L1 and register file.

Asynchronous memory copy with LDGSTS instruction vs. TMA

- **A100 Using LDGSTS instr**
  - SM
  - Tensor Core
  - Registers
  - Threads
  - SMEM
  - L1 Cache
  - Data
  - Reads
  - Global Memory
  - Addr gen by threads

- **H100 Using TMA Unit**
  - SM
  - Tensor Core
  - Registers
  - Threads
  - TMA
  - SMEM
  - L1 Cache
  - Data + TransCnt
  - Reads
  - Global Memory
  - Addr gen by TMA

**TMA unit reduces addressing overhead**

A single thread per warp issues the TMA operation

Support for different tensor layouts (1D-5D)

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
NVIDIA H100 Distributed Shared Memory

- Shared memory virtual address space distributed across the blocks of a cluster
- Load, store, and atomic operations to other SM’s shared memory

Thread block clusters and distributed shared memory (DSMEM) are leveraged via cooperative_groups API

TMA unit supports copies across thread blocks in a cluster

Asynchronous transaction barriers

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
Pageable Host Memory

- Allocated with `malloc` family of APIs
- In CPU DRAM memory when accessed from the CPU
- The Operating System might decide to move its contents to disk if not in use and bring it back on-demand as the CPU access it
- Cannot be (directly) used for DMA transfers
Pinned Host Memory

- Allocated with `mmap/mlock` and `cudaMallocHost`

- Is guaranteed to be in the CPU memory even when not accessed

- The Operating System will keep it in DRAM and promises to never move it around

- Can be used directly for DMA transfers

- Allocating too much pinned host memory might degrade overall system performance
Pinned Device Memory

- Allocated with `cudaMalloc`

- Guaranteed to be in the GPU memory even when not accessed

- The CUDA driver will keep it in DRAM and promises to never move it around

- Can be used directly for DMA transfers
Using pageable host memory

Transfer chopped in small-sized chunks

Data chunks first copied to pinned staging buffer before copied to device pinned memory

Using pinned host memory
The overhead of extra copying from host pageable to pinned host memory buffer becomes more pronounced when the link speed is higher.
Pinned Host Memory as Zero–Copy Memory

- Make pinned host memory accessible to devices connected to the CPU.

- Devices can access this memory using regular load/store instructions.

- GPU can coalesce multiple concurrent access into single PCIe/NVLink transactions to improve efficiency.

- If compute-to-memory ratio is high enough, GPU can hide long access latency to host memory.
int main(int argc, char * argv[])
{
    float * data;
    ...
    cudaHostAlloc(&data, size);
    ...
    my_gpu_kernel(data, size);
    cudaDeviceSynchronize();
    ...
    cudaFreeHost(data)
}
Breakdown of Copy/Kernel Times

- Pageable
- Pinned
- Zero-copy

github.com/cwpearson/triad-gpu-bandwidth
CUDA Unified Memory

- Allocated with `cudaMallocManaged`

- In device when accessed by the GPU and in host when accessed by CPU

- The CUDA driver moves data between host and device memory as needed

- Host and Device use the same virtual address (pointer value) for access

- No need for data transfers, but performance hints make a big difference
int main(int argc, char * argv[]) {
    float * data;
    ...
    cudaMallocManaged(&data, size);
    ...
    my_gpu_kernel(data, size);
    cudaDeviceSynchronize();
    ...
    cudaFree(data)
}
CUDA Unified Memory Performance Hints

cudaMemAdvise(void * ptr, size_t, cudaMemoryAdvise hint, int device)

// cudaMemAdviseSetReadMostly: makes a best-effort to avoid memory copies when accessing the data in read-only mode

// cudaMemAdviseSetPreferredLocation: avoids memory copies if data is accessible through other means (e.g., as zero-copy memory)

// cudaMemAdviseSetAccessedBy: avoids memory copies (if possible) when the data is accessed by the specified host/device
CUDA Unified Memory Prefetching

cudaMemPrefetchAsync(void * ptr, size_t, int device, cudaStream_t s)

“Initiates the migration of the specified memory region to the device, so no additional memory copies are required when the data is accessed
int main(int argc, char * argv[]) {
    float * data;
    ... 
    data = malloc(size);
    ...
    my_gpu_kernel(data, size);
    cudaDeviceSynchronize();
    ...
    free(data)
}
Breakdown of Kernel and Copy Time

- Breakdown of Kernel and Copy Time:
  - Breakdown of Kernel and Copy Time:
  - Breakdown of Kernel and Copy Time:
  - Breakdown of Kernel and Copy Time:
  - Breakdown of Kernel and Copy Time:
NVIDIA Grace Hopper Superchip

- CPU + GPU
  - Grace CPU + Hopper GPU
- 900 GB/s coherent interface (7x faster than PCIe Gen 5)

PUMPS+AI Summer School
September 2-6, 2022

WARP-SYNCHRONOUS PROGRAMMING
How does a GPU executes your code?

Thread-blocks are sent to Streaming Multiprocessors (SMs) for execution.

SMs accept thread-blocks as long as there are enough resources (e.g., registers, shared memory…) available.

Once a thread-block completes, the SM gets a new thread-block to execute.
How does a GPU executes your code?

- SM chops thread-blocks into fixed-size (32 in NVIDIA) sets of threads: warps

- SM executes threads within a warp in lock-step

![Diagram showing GPU execution process](image_url)
**Thread Divergence**

- What happens when threads take different control flow paths?

- The SM chooses one path and executes the threads in that path until the reconvergence point.

- The SM executes the other path until the reconvergence point.

- SM resumes converged execution.

```c
if(threadIdx.x % 1) {
  ...
}
else {
  ...
}
```
Thread Divergence in Volta

Starting in Volta the SM might decide to diverge threads at any point (e.g., on memory accesses)

The SM interleaves the execution of warps to guarantee forward progress of all threads

The SM aims to converge execution, but converged execution is not guaranteed

```c
if(threadIdx.x % 1) {
    ... 
}
else {
    ...
}
```
Warp Reduction prior Volta

__shfl(_T, int), __shfl_down(_T, int), __shfl_up(_T, int), __shfl_xor(_T, int)

Use intra-warp communication to compute reduction and broadcast

__device__
int warp_reduce_sum(int val) {
  int lane = threadIdx.x % 32;
  for(int n = 16;
      n > 0 && lane < (n << 1); n >>= 1) {
    val += __shfl_down(val, n);
  }
  __shfl(val, 0);
  return val;
}
Warp Reduction in Volta

Threads might run ahead of others, producing invalid output

Warp-synchronous operations require mask of participating threads

```c
__device__
int warp_reduce_sum(int val) {
    int lane = threadIdx.x % 32;
    unsigned mask = 0xffffffff;
    for(int n = 16;
        n > 0 && lane < (n << 1); n >>= 1) {
        val += __shfl_down_sync(mask, val, n);
        mask >>= n;
    }
    __shfl_sync(0xffffffff, val, 0);
    return val;
}
```
Warp Shuffle Functions

Built-in warp shuffle functions enable threads to share data with other threads in the same warp

- Faster than using shared memory and __syncthreads() to share across threads in the same block

Variants:

- __shfl_sync(mask, var, srcLane)
  - Direct copy from indexed lane

- __shfl_up_sync(mask, var, delta)
  - Copy from a lane with lower ID relative to caller

- __shfl_down_sync(mask, var, delta)
  - Copy from a lane with higher ID relative to caller

- __shfl_xor_sync(mask, var, laneMask)
  - Copy from a lane based on bitwise XOR of own lane ID
__global__ void reduce_kernel(float* input, float* partialSums, unsigned int N) {

    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ float input_s[BLOCK_DIM];
    input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
    __syncthreads();

    // Reduction tree in shared memory
    for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) {
        if(threadIdx.x < stride) {
            input_s[threadIdx.x] += input_s[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Reduction tree with shuffle instructions
    float sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];
        for(unsigned int stride = WARP_SIZE/2; stride > 0; stride /= 2) {
            sum += __shfl_down_sync(0xffffffff, sum, stride);
        }
    }

    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = sum;
    }
}
__global__ void reduce_kernel(int* input, int* partialSums, unsigned int N) {

    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ int input_s[BLOCK_DIM];
    input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
    __syncthreads();

    // Reduction tree in shared memory
    for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) {
        if(threadIdx.x < stride) {
            input_s[threadIdx.x] += input_s[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Reduction with warp reduce instruction
    int sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];
    }
    // warp reduce intrinsic for cc 8.0 or higher
    sum = __reduce_add_sync(0xffffffff, sum);

    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = sum;
    }
}
Ampere (cc 8.x) adds native support for warp-wide reduction operations

**B.21. Warp Reduce Functions**

The __reduce_sync(unsigned mask, T value) intrinsics perform a reduction operation on the data provided in value after synchronizing threads named in mask. T can be unsigned or signed for [add, min, max] and unsigned only for [and, or, xor] operations.

Supported by devices of compute capability 8.x or higher.

**B.21.1. Synopsis**

```
// add/min/max
unsigned __reduce_add_sync(unsigned mask, unsigned value);
unsigned __reduce_min_sync(unsigned mask, unsigned value);
unsigned __reduce_max_sync(unsigned mask, unsigned value);
int __reduce_add_sync(unsigned mask, int value);
int __reduce_min_sync(unsigned mask, int value);
int __reduce_max_sync(unsigned mask, int value);

// and/or/xor
unsigned __reduce_and_sync(unsigned mask, unsigned value);
unsigned __reduce_or_sync(unsigned mask, unsigned value);
unsigned __reduce_xor_sync(unsigned mask, unsigned value);
```

**B.21.2. Description**

**__reduce_add_sync, __reduce_min_sync, __reduce_max_sync**

Returns the result of applying an arithmetic add, min, or max reduction operation on the values provided in value by each thread named in mask.

**__reduce_and_sync, __reduce_or_sync, __reduce_xor_sync**

Returns the result of applying a logical AND, OR, or XOR reduction operation on the values provided in value by each thread named in mask.

The mask indicates the threads participating in the call. A bit, representing the thread’s lane id, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware. All non-exited threads named in mask must execute the same intrinsic with the same mask, or the result is undefined.

Other Warp-synchronous Primitives

```c
__syncwarp(unsigned)
```

Forces the reconvergence of the threads in the mask

```c
__activemask()
```

Returns the mask of converged threads

```c
__all_sync(unsigned, bool) and __any_sync(unsigned, bool)
```

Returns true if all or any of the participating threads pass true
Other Warp-synchronous Primitives

```c
__ballot_sync(unsigned, bool)
```

Returns the mask of threads that passed true

```c
__match_all_sync(unsigned, _T)
```

Returns true if all participating threads pass the same value

```c
__match_any_sync(unsigned, _T)
```

Returns the mask of participating threads passing the same value
Coalesced Atomic Operations

**Identify threads operating on the same atomic and use a reduction**

```c
int atomic_add(int * ptr, int value) {
    unsigned active_mask = __activemask();
    unsigned active_mask = __match_any_sync(active_mask, ptr);

    int value = reduce_warp(active_mask, value);
    if(__ffs(active_mask) - 1) == lane) {
        value = atomicAdd(ptr, value);
    }

    value = __shfl_sync(active_mask, value, __ffs(active_mask) - 1);
    return value;
}
```