CUTLASS: CUDA TEMPLATE LIBRARY FOR DENSE LINEAR ALGEBRA AT ALL LEVELS AND SCALES

Andrew Kerr, Duane Merrill, Julien Demouth, John Tran, Naila Farooqui, Markus Tavenrath, Vince Schuster, Eddie Gornish, Jerry Zheng, Bageshri Sathe
OUTLINE

CUTLASS Introduction and Roadmap

Efficient Linear Algebra Computations on GPUs

CUTLASS Deep Dive
MOTIVATION

Productivity Challenges in Deep Learning

Problem:

Multiplicity of Algorithms and Data Types
  • GEMM, Convolution, Back propagation
  • Mixed precision arithmetic
Kernels specialized for layout and problem size
  • NT, TN, NCHW, NHWC
Kernel Fusion
  • Custom operations composed with GEMM and convolution

Solution:

Template Library for Linear Algebra Computations in CUDA C++
  • Thread-wide, warp-wide, block-wide, device-wide
Data movement and computation primitives
  • Iterators, matrix fragments, matrix computations
Inspired by CUB
PREVIOUSLY: CUTLASS 0.1

Preview Release - December 2017

Template-oriented Implementation

• Github: https://github.com/NVIDIA/cutlass/releases/tag/v0.1.0
• Parallel For All Blog Post: https://devblogs.nvidia.com/parallelforall/cutlass-linear-algebra-cuda/

Complete implementations

• GEMM: Floating point, Integer-valued, Volta TensorCores
SOON: CUTFIASS 1.0

April 2018

Core API

• **Shapes and tiles:** structured layout definitions and tile sizes
• **Fragments and iterators:** collective operations for efficient and composable data movement
• **Accumulator tiles and epilogues:** matrix math operations and efficient block-level reductions

Complete implementations

• **GEMM:** Floating point, Integer, Volta TensorCores

Open Source (3-clause BSD License) [https://github.com/NVIDIA/cutlass](https://github.com/NVIDIA/cutlass)
DESIGN OBJECTIVES
Span the Design Space with Generic Programming

CUDA C++ templates for composable algorithms

Performance: Implement efficient dense linear algebra kernels

Structured, reusable components: flexibility and productivity
### CUTLASS PERFORMANCE

**CUTLASS GEMM Performance**  
(M=10240, N=4096, K=4096) Quadro V100

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Performance (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HGEMM_TT</td>
<td>94%</td>
</tr>
<tr>
<td>HGEMM_TN</td>
<td>95%</td>
</tr>
<tr>
<td>HGEMM_NN</td>
<td>95%</td>
</tr>
<tr>
<td>HGEMM_NT</td>
<td>90%</td>
</tr>
<tr>
<td>DGEMM_TT</td>
<td>99%</td>
</tr>
<tr>
<td>DGEMM_TN</td>
<td>100%</td>
</tr>
<tr>
<td>DGEMM_NN</td>
<td>97%</td>
</tr>
<tr>
<td>DGEMM_NT</td>
<td>89%</td>
</tr>
<tr>
<td>SGEMM_TT</td>
<td>99%</td>
</tr>
<tr>
<td>SGEMM_TN</td>
<td>92%</td>
</tr>
<tr>
<td>SGEMM_NN</td>
<td>94%</td>
</tr>
<tr>
<td>SGEMM_NT</td>
<td>95%</td>
</tr>
</tbody>
</table>

Performance relative to cuBLAS
## IMPLEMENTED COMPUTATIONS

**CUTLASS v1.0**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEMM</td>
<td>float</td>
<td>float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>DGEMM</td>
<td>double</td>
<td>double</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>HGEMM</td>
<td>half</td>
<td>half</td>
<td>half</td>
<td>half</td>
</tr>
<tr>
<td>IGEMM</td>
<td>int8_t</td>
<td>int8_t</td>
<td>int8_t</td>
<td>int32_t</td>
</tr>
<tr>
<td></td>
<td>int8_t</td>
<td>int8_t</td>
<td>float</td>
<td>int32_t</td>
</tr>
<tr>
<td>WMMA GEMM</td>
<td>half</td>
<td>half</td>
<td>half</td>
<td>half</td>
</tr>
<tr>
<td></td>
<td>half</td>
<td>half</td>
<td>float</td>
<td>float</td>
</tr>
</tbody>
</table>
GEMM TEMPLATE KERNEL

CUTLASS provides building blocks for efficient device-side code

- Helpers simplify common cases

```
// CUTLASS GEMM kernel
//
template <typename Gemm>
__global__ void gemm_kernel(typename Gemm::Params params) {
    // Declare shared memory
    __shared__ typename Gemm::SharedStorage shared_storage;

    // Construct the GEMM object with cleared accumulators
    Gemm gemm(params);

    // Compute the matrix multiply-accumulate
    gemm.multiply_add(shared_storage.mainloop);

    // Update output memory efficiently
    gemm.update(shared_storage.epilogue);
}
```

```
// Specialization for single-precision
//
typedef cutlass::gemm::SgemmTraits<
cutlass::MatrixLayout::kColumnMajor,
cutlass::MatrixLayout::kRowMajor,
cutlass::Shape<8, 128, 128>
> SgemmTraits;

// Simplified kernel launch
Gemm<SgemmTraits>::launch(params);
```
EFFICIENT LINEAR ALGEBRA COMPUTATIONS ON GPUS
GENERAL MATRIX PRODUCT

Basic definition

General matrix product

\[ C = \alpha \text{op}(A) \ast \text{op}(B) + \beta C \]

\( C \) is \( M \)-by-\( N \), \( \text{op}(A) \) is \( M \)-by-\( K \), \( \text{op}(B) \) is \( K \)-by-\( N \)

Compute independent dot products

```c
// Independent dot products
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

Inefficient due to large working sets to hold parts of \( A \) and \( B \)
GENERAL MATRIX PRODUCT

Accumulated outer products

General matrix product

\[ C = \alpha \text{op}(A) \ast \text{op}(B) + \beta \ C \]

\( C \) is \( M \)-by-\( N \), \( \text{op}(A) \) is \( M \)-by-\( K \), \( \text{op}(B) \) is \( K \)-by-\( N \)

Compute independent dot products

// Independent dot products
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];

Permute loop nests

// Accumulated outer products
for (int k = 0; k < K; ++k)
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            C[i][j] += A[i][k] * B[k][j];

Load elements of \( A \) and \( B \) exactly once
GENERAL MATRIX PRODUCT

Computing matrix product one block at a time

Partition the loop nest into \textit{blocks} along each dimension

\begin{itemize}
  \item Partition into $Mtile$-by-$Ntile$ independent matrix products
  \item Compute each product by accumulating $Mtile$-by-$Ntile$-by-$Ktile$ matrix products
\end{itemize}

\begin{verbatim}
for (int mb = 0; mb < M; mb += Mtile)
  for (int nb = 0; nb < N; nb += Ntile)
    for (int kb = 0; kb < K; kb += Ktile)
      {
        // compute $Mtile$-by-$Ntile$-by-$Ktile$ matrix product
        for (int k = 0; k < Ktile; ++k)
          for (int i = 0; i < Mtile; ++i)
            for (int j = 0; j < Ntile; ++j)
              {
                int row = mb + i;
                int col = nb + j;

                C[row][col] +=
                A[row][kb + k] * B[kb + k][col];
              }
      }
\end{verbatim}
**BLOCKED GEMM IN CUDA**

Parallelism Among CUDA Thread Blocks

Launch a CUDA kernel grid

- Assign CUDA thread blocks to each partition of the output matrix

CUDA thread blocks compute $M_{tile}$-by-$N_{tile}$-by-$K$ matrix product in parallel

- Iterate over $K$ dimension in steps, performing an accumulated matrix product

```c
for (int mb = 0; mb < M; mb += M_{tile})
    for (int nb = 0; nb < N; nb += N_{tile})
        for (int kb = 0; kb < K; kb += K_{tile})
            { .. compute $M_{tile}$ by $N_{tile}$ by $K_{tile}$ GEMM }
```

by each CUDA thread block
Decompose thread block into warp-level tiles

- Load A and B operands into Shared Memory (reuse)
- C matrix distributed among warps

Each warp computes an independent matrix product

```c
for (int kb = 0; kb < K; kb += Ktile) {
    .. load A and B tiles to shared memory
    for (int m = 0; m < Mtile; m += warp_m)
        for (int n = 0; n < Ntile; n += warp_n)
            for (int k = 0; k < Ktile; k += warp_k)
                .. compute warp_m by warp_n by warp_k GEMM
}
```

by each CUDA warp
WARP-LEVEL TILE STRUCTURE

Warp-level matrix product

Warps perform an accumulated matrix product

- Load $A$ and $B$ operands from SMEM into registers
- $C$ matrix held in registers of participating threads

Shared Memory layout is $K$-strided for efficient loads

```c
for (int k = 0; k < Ktile; k += warp_k)
{
    // load A tile from SMEM into registers
    // load B tile from SMEM into registers
    for (int tm = 0; tm < warp_m; tm += thread_m)
        for (int tn = 0; tn < warp_n; tn += thread_n)
            // compute thread_m by thread_n by thread_k GEMM
}
```

by each CUDA thread
THREAD-LEVEL TILE STRUCTURE

Parallelism within a thread

Threads compute accumulated matrix product

- $A$, $B$, and $C$ held in registers

Opportunity for data reuse:

- $O(M \times N)$ computations on $O(M+N)$ elements

```c
for (int m = 0; m < thread_m; ++m)
    for (int n = 0; n < thread_n; ++n)
        for (int k = 0; k < thread_k; ++k)
            C[m][n] += A[m][k] * B[n][k];
```

Fused multiply-accumulate instructions
COMPLETE GEMM HIERARCHY

Data reuse at each level of the memory hierarchy
CUTLASS DEEP DIVE
CUTLASS DESIGN PATTERNS

Design patterns and template concepts in CUTLASS

**Templates:** generic programming and compile-time optimizations

**Traits:** describes properties, types, and functors used to specialize CUTLASS concepts

**Params:** structure containing parameters and precomputed values; passed to kernel as POD

**Vectorized Memory Accesses:** load and store as 32b, 64b, or 128b vectors

**Shape<>:** describes size of a 4D vector quantity

**TileTraits<>:** describes a 4D block of elements in memory

**Fragment<>:** partitioning of a tile across a collection of threads

**TileIterator<>:** loads a tile by a collection of threads; result is held in Fragment
GEMM HIERARCHY: THREAD BLOCKS

Streaming efficiently to shared memory
LOADING A TILE INTO FRAGMENTS
Abstractions for efficient data transfer

Fragment: object containing each thread’s partition of a tile

Example: strip-mining a 16-by-16 tile across 32 threads, loading as 2-vector

Fragment<float, 8>
similar to std::array<float, 8>
Tile Traits: tile dimensions, fragment size, access pitch, and initial offset function

```cpp
// Concept specifying traits of a tile in memory
struct TileTraits {

  // Shape of the tile in memory
  typedef Shape<1, 16, 8, 2> Tile;

  // Number of accesses performed
  typedef Shape<1, 4, 1, 1> Iterations;

  // Number of steps along each dimension between accesses
  typedef Shape<1, 4, 1, 1> Steps;

  // Function to compute each thread’s initial offset in the tile
  __host__ __device__
  static Coord<4> thread_offset() const {
    return make_Coord(0, threadIdx.x / 8, threadIdx.x % 8, 0);
  }
};
```
**TILE ITERATORS**

Abstraction for accessing tiles in memory

Tile Iterator: owns pointer and strides

// Construct load and store iterators from base pointers and strides
TileLoadIterator<TileTraits, float, MemorySpace::kGlobal> gmem_load(gmem_ptr, gmem_leading_dim);
TileStoreIterator<TileTraits, float, MemorySpace::kShared> smem_store(smem_ptr, kSmemPitch);

// Load a fragment from global memory and store to shared memory
Fragment frag;
iterator_load_post_increment(gmem_load, frag);
iterator_store(smem_store, frag);
ARBITRARY MATRIX DIMENSIONS

Using guard predicates with iterators

Iterators accept predicate vectors when loading or storing tiles

- One predicate per memory access

GEMM computes guard predicates before entering mainloop

- Predicates updated once, prior to final Ktile iteration

```cpp
// Construct a tile load iterator with bounds
TileLoadIterator gmem_load(params, make_Coord(1, K, M));

// Initialize predicate vector with the tile load iterator
typename TileLoadIterator::PredicateVector predicates;
gmem_load.initialize_predicates(threadblock_offset, predicates.begin());

// Load tiles while iterating over K dimension
iterator_load_post_increment(gmem_load, frag, predicates.const_begin());
...

// Update predicates and load final tile
gmem_load.residue(K_remainder);
iterator_load(gmem_load, frag, predicates.const_begin());
```
GEMM HIERARCHY: WARP TILES

Loading multiplicands into registers
SHARED MEMORY TO REGISTERS

Load $A$ and $B$ fragments from Shared Memory with iterators

- **SMEM to RF**: must load data faster than math throughput

Tile iterator traits determined by math instruction

Typical warp-tile fragment sizes:
- **SGEMM, DGEMM**: 64-by-32-by-1
- **HGEMM**: 128-by-32-by-1
- **IGEMM**: 64-by-32-by-4
- **WMMA GEMM**: 64-by-32-by-16
GEMM HIERARCHY: CUDA CORES

Actually doing the math
REGISTERS TO CUDA CORES

Compute matrix multiply-accumulate on fragments held in registers

```cpp
// Perform thread-level matrix multiply-accumulate
template <
  typename Shape,
  typename ScalarA,
  typename ScalarB,
  typename ScalarC
>
struct GemmMultiplyAdd {

  /// Multiply: D = A*B + C
  inline __device__ void multiply_add(
    Fragment<ScalarA, Shape::kW> const & A,
    Fragment<ScalarB, Shape::kH> const & B,
    Accumulators const & C,
    Accumulators & D)
  {
    for (int j = 0; j < Shape::kH; ++j) {
      for (int i = 0; i < Shape::kW; ++i) {
        D.scalars[j * Shape::kW + i] =
          // multiply
          A.scalars[i] * B.scalars[j] +
          // add
          C.scalars[j * Shape::kW + i];
      }
    }
  }
};
```
EXAMPLE: VOLTA TENSOR CORES

Targeting the CUDA WMMA API

WMMA: Warp-synchronous Matrix Multiply-Accumulate
- API for issuing operations to Volta Tensor Cores

```cpp
/// Perform warp-level multiply-accumulate using WMMA API
/// Targeting the CUDA WMMA API
>

struct WmmaMultiplyAdd {
    /// Data type of accumulator
typename ScalarC,
    /// Shape of warp-level accumulator tile
typename WarpTile,
    /// Shape of one WMMA operation - e.g. 16x16x16
typename WmmaTile
>
    template <
        /// Data type of accumulator
typename ScalarC,
        /// Shape of warp-level accumulator tile
typename WarpTile,
        /// Shape of one WMMA operation - e.g. 16x16x16
typename WmmaTile>
    struct WmmaMultiplyAdd {
        /// Compute number of WMMA operations
typedef typename ShapeDiv<WarpTile, WmmaTile>::Shape
        /// Multiply: D = A*B + C
inline _device_ void multiply_add(
            FragmentA const & A,
            FragmentB const & B,
            FragmentC const & C,
            FragmentD & D) {
        for (int n = 0; n < Shape::kH; ++n) {
            for (int m = 0; m < Shape::kW; ++m) {
                // WMMA API to invoke Tensor Cores
                nvcuda::wmma::mma_sync(
                    D.elements[n][m],
                    A.elements[k][m],
                    B.elements[k][n],
                    C.elements[n][m]
                );
            }
        }
    }
};
```
EXAMPLE: IGEMM

Mixed-precision Integer-valued GEMM

DP4A instruction computes 4-element dot product

- $A$ and $B$ are packed vectors of 8-bit integers
- Accumulator is 32-bit signed integer

```cpp
/// Perform $M$-by-$N$-by-$4$ matrix product using DP4A
template <typename Shape>
struct IgemmMultiplyAdd<Shape, int8_t, int8_t, int>

/// Multiply: $d = a*b + c$
inline device void multiply_add(
  Fragment<int8_t, Shape::kW * 4> const & A,
  Fragment<int8_t, Shape::kH * 4> const & B,
  Accumulators const & C,
  Accumulators & D)
{
  int const* a_int = reinterpret_cast<int const*>(&A.scalars[0]);
  int const* b_int = reinterpret_cast<int const*>(&B.scalars[0]);
  // Perform $M$-by-$N$-by-$4$ matrix product using DP4A
  for (int j = 0; j < Shape::kH; ++j) {
    for (int i = 0; i < Shape::kW; ++i) {
      // Inline PTX to issue DP4A instruction
      asm volatile("dp4a.s32.s32 %0, %1, %2, %3;":
        "=r"(D.scalars[j * Shape::kW + i])
        : "r"(a_int[i]),
        "r"(b_int[j]),
        "r"(C.scalars[j * Shape::kW + i]));
    }
  }
}
```
EXAMPLE: IGEMM

Interleaved data layouts for efficient streaming from Shared Memory

DP4A requires operands to be contiguous along $K$ dimension

- Efficient fragment loading requires $K$-strided layout in Shared Memory
- **Solution:** adopt a hybrid SMEM layout

```
int8_t[4][4]
```

![Diagram showing data layout and streaming from Shared Memory](image)
GEMM HIERARCHY: TRANSFORMING FRAGMENTS
Permute fragments before storing to shared memory

PTX ISA: prmt
Accumulator tiles typically don’t match output matrix

- Element-wise operation: $C = \alpha AB + \beta C$
- Type Conversion: scale, convert, and pack into vectors
- Layout: $C$ matrix is contiguous
SPATIALLY INTERLEAVED ACCUMULATORS

Warp tile need not be contiguous
GEMM EPILOGUE

Restructuring accumulators, elementwise operators, and updating global memory

wmma::store_matrix_sync()
KERNEL FUSION

Custom element-wise operations during epilogue

Matrix product may be combined with arbitrary functions

- **Element-wise operators**: Scaling, bias, activation functions
- **Data type conversion**: F32→F16, Int32→Int8
- **Matrix update operations**: reductions across thread blocks
COMPLETE* GEMM DATA FLOW
Embodied by CUTLASS CUDA templates

* Mostly. Not depicted: software pipelining, double-buffering, and more. Read the code. ☺
CONCLUSION

CUTLASS: CUDA C++ Template Library

CUTLASS is an Open Source Project for implementing Deep Learning computations on GPUs
- https://github.com/nvidia/cutlass (3-clause BSD License)
- V1.0: April 2018

CUTLASS is efficient: >90% cuBLAS performance

Generic programming techniques span Deep Learning design space
- Hierarchical decomposition of GEMM
- Data movement primitives
- Mixed-precision and Volta Tensor Cores

CUTLASS enables developers to compose custom Deep Learning CUDA kernels
QUESTIONS?

CUTLASS:  https://github.com/nvidia/cutlass

We welcome your feedback!