High Performance Linear Algebra

Lecture 15: Some hints on BLAS on GPUs

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

Thursday 5, 2026 — 16.00:18.00

Dipartimento
di Matematica
Università di Pisa

HPLA up to now and today's plan

1 GPU BLAS libraries

Up to now, we have seen:

- The design principles of the BLAS library,
- The implementation of the BLAS library on CPUs and shared memory systems,
- Some performance considerations on CPU BLAS implementations,
- The implementation of the BLAS library on distributed memory systems.
- Some performance considerations on distributed memory BLAS implementations.
- LAPACK and ScaLAPACK libraries for dense linear algebra.

Today we will look at some implementations of the BLAS library on GPUs.

Table of Contents

2 GPGPU Computing and Dense Linear Algebra

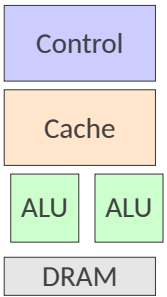
GPGPU Computing

- **GPGPU** stands for **G**eneral-**P**urpose computing on **G**raphics **P**rocessing **U**nits.
- Originally designed for image rendering, GPUs have evolved into highly parallel, multi-threaded, many-core processors with tremendous computational power and very high memory bandwidth.
- Why leverage GPUs for scientific computing?
  - High arithmetic intensity.
  - Massive parallelism (thousands of cores).
  - Energy efficiency (FLOPS per Watt).
- Dense Linear Algebra (DLA) is a prime candidate for GPU acceleration due to its regular memory access patterns and high computational density.

CPU vs. GPU Architecture

CPU (Latency Oriented)

- Optimized for serial performance.
- Complex control logic.
- Large caches to minimize latency.
- Fewer, powerful cores.

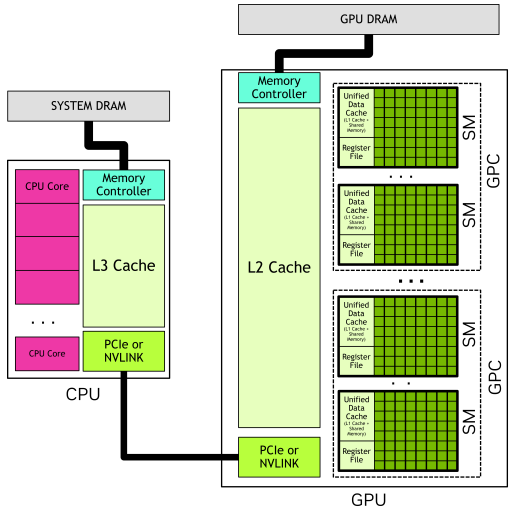


GPU (Throughput Oriented)

- Optimized for parallel performance.
- Simple control logic.
- Small caches (latency hidden by thread switching).
- Many simpler cores (SIMT).

CPU vs. GPU Architecture

- **High-Level View**: GPU = collection of **Streaming Multiprocessors (SMs)**.
- **Organization**: SMs are grouped into Graphics Processing Clusters (GPCs).
- **Inside an SM**:
  - Register file.
  - Unified data cache (L1 cache + Shared Memory).
  - Functional units (CUDA cores, Tensor cores).
- **Flexibility**: The split between L1 and Shared Memory is configurable at runtime.



Programming Models for GPUs

To utilize GPUs for linear algebra, we need specific programming models:

CUDA (Compute Unified Device Architecture) Proprietary NVIDIA platform. The de-facto standard for HPC on NVIDIA GPUs. Provides low-level control and high performance.

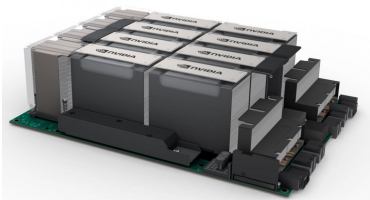
HIP (Heterogeneous-Compute Interface for Portability) AMD's answer to CUDA, allowing code to run on AMD hardware.

OpenCL / SYCL / OneAPI Open standards for cross-platform parallel programming.

OpenACC / OpenMP Directive-based approaches (pragmas) to offload computations to accelerators without rewriting the entire codebase.

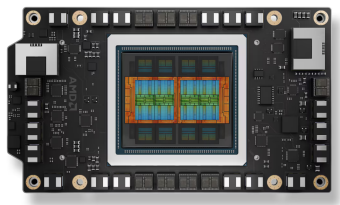
Most vendor-provided BLAS libraries (cuBLAS, rocBLAS) are highly optimized using the native models (CUDA/HIP).

How much does it costs?



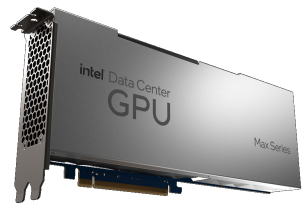
NVIDIA HGX200

- 141GB of HBM3e
- 4.8TB/s of bandwidth
- 4 petaFLOPS for FP8
- $ ∼ 300.000 $



AMD Instinct™ MI355X GPUs

- 288GB of HBM3e
- 8 TB/s of bandwidth
- 5 petaFLOPS for FP8
- $ ∼220.000 €



Intel® GPU Max 1550

- 128 GB of HBM2e
- 3.28 TB/s of bandwidth
- 52.43 TFLOPS for FP8
- $ ∼8.550 €

The CUDA programming model assumes a heterogeneous system:

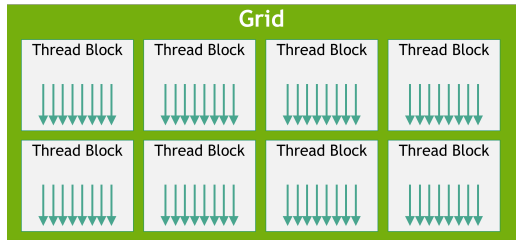
- **Host**: The CPU and its memory (system memory).
- **Device**: The GPU and its own memory.

Typically, a CUDA program flows as follows:

1. **Copy data** from Host memory to Device memory.
2. **Launch Kernel**: The CPU instructs the GPU to execute a function (kernel) on the data.
3. **Execute**: The GPU executes the kernel in parallel across many threads.
4. **Copy results** from Device memory back to Host memory.

Thread blocks and grids

- **High count:** Kernels launch millions of threads, organized into **Blocks**.
- **Structure:** blocks form a **Grid**.
  - Blocks in a grid have the same size.
  - Can be 1D, 2D, or 3D for easy mapping to data.
- **Identity:** Threads use built-in variables to find their coordinates in the Block/Grid.

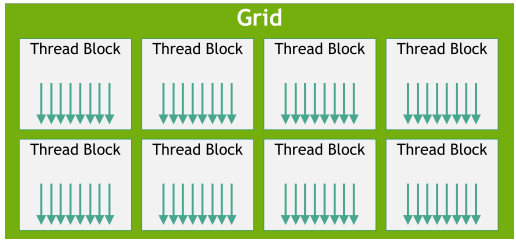


The CUDA programming model enables arbitrarily large grids to run on GPUs of any size. To achieve this, it requires that there be **no data dependencies** between threads in different thread blocks.

Thread blocks and grids

- **Hardware Mapping:**
  - A block executes on a single **SM**.
  - Enables fast synch and **Shared Memory** usage within a block.
  - Grids scale to millions of blocks, automatically scheduled on available SMs.

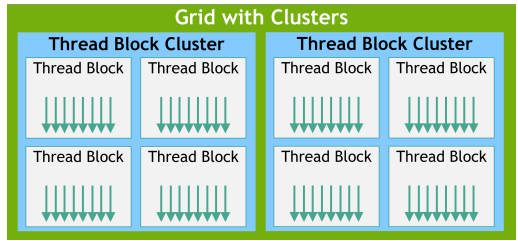
Threads within a block run on the same SM, while different thread blocks are scheduled among available SMs in any order. This allows the execution to be parallel or serial, ensuring scalability across different hardware.



The CUDA programming model enables arbitrarily large grids to run on GPUs of any size. To achieve this, it requires that there be **no data dependencies** between threads in different thread blocks.

Thread blocks and grids

- **Clusters (CC $\geq$ 9.0):** An optional hierarchy level grouping thread blocks.
- **Execution:** Guaranteed to run on a single **GPC** (Graphics Processing Cluster).
- **Benefits:**
  - Synchronization between blocks in the same cluster.
  - **Distributed Shared Memory:** Threads can access the shared memory of all blocks in the cluster.



The CUDA programming model enables arbitrarily large grids to run on GPUs of any size. To achieve this, it requires that there be **no data dependencies** between threads in different thread blocks.

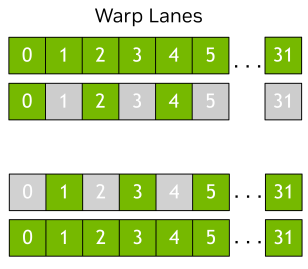
Warps and SIMT

- **Warps**:
  - Threads within a block are grouped into bundles of **32 threads** called *warps*.
  - Warps are the fundamental unit of scheduling on an SM.
- **SIMT (Single-Instruction, Multiple-Threads)**:
  - All threads in a warp execute the **same instruction** at the same time.
  - Each thread has its own instruction address counter and register state.
  - *Lock-step execution*: Ideally, all 32 threads progress together.
- **Recommendation**:
  - Configure thread block sizes to be multiples of 32.
  - If not, the last warp will have inactive lanes, wasting computational resources.

Warp Divergence

- Threads in a warp can follow different execution paths (e.g., `if-else` statements).
- **Divergence**: If threads diverge, the warp serially executes each branch path.
- Threads not on the current path are **masked off** (inactive).
- **Impact**: Significantly reduces parallel efficiency (active threads < 32).
- **Optimization**: Maximize utilization by ensuring threads in a warp follow the same control flow.

```
if(threadIdx.x%2 == 0)
   {a = r(t); }
else
   {a = q(t); }
y = f(a);
```



GPU Memory Model Overview

Modern GPUs utilize a complex memory hierarchy to balance bandwidth and latency. We distinguish between:

- **Off-Chip Memory (DRAM)**:
  - **Global Memory**: The large DRAM attached to the GPU. Accessible by all SMs. High latency, high bandwidth.
  - **System Memory**: DRAM attached to the CPU (Host).
- **On-Chip Memory**:
  - **Registers**: Fastest memory, private to a thread.
  - **Shared Memory**: Programmable cache shared within a Thread Block (or Cluster).
  - **Caches**: L1 (per SM) and L2 (device-wide).

Unified Addressing: CPU and GPU share a single virtual memory space, allowing the unique identification of memory locations across devices.

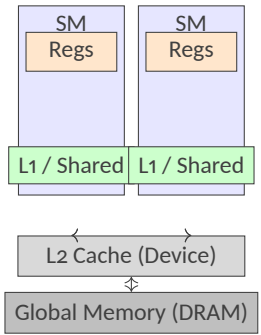
On-Chip Memory Details

Registers

- Stores thread-local variables.
- Allocation is per-thread; total usage determines *occupancy* (how many blocks fit on an SM).

Shared Memory

- Visible to all threads in a Block (or Cluster on H100+).
- Used for inter-thread communication and data reuse.
- Physically shares storage with L1 Cache; the split is often configurable.



The CUDA Platform Overview

The NVIDIA CUDA platform enables heterogeneous computing through a combination of hardware and software components.

Key Components:

- **Compute Capability (CC):**
  - Version number (X.Y) indicating supported features and hardware parameters.
  - Corresponds to the Streaming Multiprocessor (SM) version (e.g., CC 9.0 $\rightarrow$ `sm_90`).
- **NVIDIA Driver:**
  - acts as the ”OS” of the GPU.
  - Foundational; required for all GPU uses (CUDA, Vulkan, Direct3D).
- **CUDA Toolkit:**
  - Suite of libraries, headers, tools (e.g., `nvcc`), and the **CUDA Runtime**.

CUDA APIs: Runtime vs. Driver

CUDA offers two main APIs for application development:

CUDA Runtime API

- High-level API.
- Handles common tasks (memory allocation, data transfer, kernel launching) easily.
- Language extensions (e.g., <<<...>>> syntax).
- Used in most CUDA applications.

CUDA Driver API

- Low-level API exposed directly by the driver.
- Grants finer control (e.g., context management).
- More verbose; conceptually similar to OpenCL.

- The Runtime API is implemented *on top* of the Driver API.
- Applications can mix both APIs (interoperability).

Parallel Thread Execution (PTX)

- **What is PTX?**
  - A virtual Instruction Set Architecture (ISA).
  - An intermediate assembly language that abstracts the physical hardware.
- **Role in Compilation:**
  - High-level code (C++/Fortran) is compiled into PTX.
  - The graphics driver *Just-In-Time (JIT)* compiles PTX into machine code (SASS) valid for the specific GPU installed.
- **Benefit:**
  - *Forward compatibility*: Code compiled to PTX years ago can often run on new GPUs because the driver translates the virtual instructions to the new architecture.

Binary Compatibility

NVIDIA GPUs guarantee binary compatibility under specific conditions:

- **Within Major Version:**
  - Binary code (`cubin`, e.g., `sm_86`) can run on GPUs with the *same major version* and equal or higher minor version.
  - *Example:* Code compiled for `sm_86` works on `sm_86` and `sm_89`, but **not** on `sm_80` (minor version too low).
- **Between Major Versions:**
  - Binaries are **not** compatible across major versions.
  - *Example:* Code for `sm_80` will not run on a Hopper GPU (`sm_90`).

Note: Binary compatibility applies only to binaries generated by official NVIDIA tools (e.g., `nvcc`).

PTX Compatibility & Forward Compatibility

To support future hardware, GPU code can be embedded as PTX (virtual assembly) rather than just binary SASS.

- **Mechanism:**
  - Application stores PTX for a specific virtual architecture (e.g., `compute_80`).
  - At runtime, the driver **Just-In-Time (JIT)** compiles this PTX into binary code for the detected GPU.
- **Requirement:** The PTX version must be $\leq$ the GPU's compute capability.
- **Benefit:**
  - *Forward Compatibility*: An app compiled today for `compute_80` can run on a future architecture (e.g., `sm_120`) without recompilation.

Just-in-Time (JIT) Compilation

- **Process**: The device driver translates loaded PTX into native binary code at application load time.
- **Trade-offs**:
  - 👎 Increases application load / startup time.
  - 👍 Allows code to run on GPUs that didn't exist when the app was built.
  - 👍 Benefits from newer compiler optimizations in updated drivers.
- **Compute Cache**:
  - The driver caches generated binaries to avoid recompiling on subsequent runs.
  - Cache is invalidated upon driver updates.
- **NVRTC**: A runtime compilation library that allows compiling CUDA C++ source code directly to PTX at runtime, offering even more flexibility.

A Simple CUDA hell-world Program


```cpp
#include <iostream>
__global__ void helloFromGPU() {
    printf("Hello World from GPU!\n");
}
int main() {
    // Launch kernel with 1 block of 1 thread
    helloFromGPU<<<1, 1>>>();
    // Wait for GPU to finish
    cudaDeviceSynchronize();
    return 0;
}
```

- The `__global__` qualifier indicates a kernel function executed on the GPU.
- The `<<<1, 1>>>` syntax launches the kernel with 1 block of 1 thread.
- `cudaDeviceSynchronize()` ensures the CPU waits for the GPU before exiting.

Compilation

To **compile** the CUDA program, use the NVIDIA CUDA Compiler `nvcc`:

```
nvcc -o hello_cuda hello_cuda.cu
```

This command generates an executable named `hello_cuda`.

We can specify the target GPU architecture using the `-arch` flag:

```
nvcc -arch=sm_89 -o hello_cuda hello_cuda.cu
```

This compiles the code for GPUs with Compute Capability 8.9 (The NVIDIA GeForce RTX 4060 on my laptop).

If we run the program, we should see:

```
Hello World from GPU!
```

Compilation with CMake

To compile CUDA code with CMake, we need to enable CUDA support in our `CMakeLists.txt` file:

```
cmake_minimum_required(VERSION 3.18)
project(HelloCUDA LANGUAGES CXX CUDA)
add_executable(hello_cuda hello_cuda.cu)
set_target_properties(hello_cuda PROPERTIES
    CUDA_ARCHITECTURES "89"
)
```

- CUDA is treated as a first-class language in CMake.
- The `set_target_properties(target CUDA_ARCHITECTURES)` property specifies the target GPU architectures.

Implementing DAXPY with CUDA

The DAXPY operation computes $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$ for vectors $\mathbf{x}$ and $\mathbf{y}$ and scalar α. Here is a simple CUDA implementation:

```
__global__ void daxpy(int n, double alpha, const double *x, double *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        y[i] += alpha * x[i];
    }
}
```

- Each thread computes one element of the result.
- The thread index i is calculated using block and thread indices.
- A boundary check ensures we do not access out-of-bounds memory.

Block and Thread Indexing

In the DAXPY kernel, each thread computes a single element of the output vector $\mathbf{y}$. The thread index i is calculated as:

$$i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

blockIdx.x The index of the current block in the grid.

blockDim.x The number of threads per block.

threadIdx.x The index of the thread within its block.

This calculation allows us to **uniquely identify each thread** across the entire grid, enabling parallel computation of the DAXPY operation.

Launching the DAXPY Kernel

To launch the DAXPY kernel from the host (CPU) code, we need to allocate memory on the GPU, copy data, and invoke the kernel:

```
// Host code
int n = 1<<20; // Vector size
double alpha = 2.0;
double *h_x = (double*)malloc(n * sizeof(double));
double *h_y = (double*)malloc(n * sizeof(double));
// Initialize h_x and h_y...
double *d_x, *d_y;
cudaMalloc(&d_x, n * sizeof(double));
cudaMalloc(&d_y, n * sizeof(double));
cudaMemcpy(d_x, h_x, n * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, n * sizeof(double), cudaMemcpyHostToDevice);
```

Launching the DAXPY Kernel


```
// Launch kernel
int blockSize = 256;
int numBlocks = (n + blockSize - 1) / blockSize;
daxpy<<<numBlocks, blockSize>>>(n, alpha, d_x, d_y);
// Copy result back to host
cudaMemcpy(h_y, d_y, n * sizeof(double), cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_x);
cudaFree(d_y);
```

- We allocate device memory using `cudaMalloc` and copy data with `cudaMemcpy`.
- The kernel is launched with a calculated number of blocks and threads per block.
- Finally, we copy the result back to the host and free device memory.

Executing a Kernel

We execute a kernel on the GPU using the triple angle bracket syntax <<<...>>>:

```
daxpy<<<numBlocks, blockSize>>>(n, alpha, d_x, d_y);
```

Here:

- `numBlocks` specifies the number of thread blocks in the grid.
- `blockSize` specifies the number of threads per block.

Example Calculation:

- The `int n=1<<20;` sets the vector size to $2^{20} = 1,048,576$.
- For a vector of size $n = 1,000,000$ and a block size of 256:
- Number of blocks = $\lceil \frac{1,000,000}{256} \rceil = 3907$.

What performances do we get out of this?

- The performance of our simple DAXPY implementation will depend on several factors:

  Memory bandwidth The speed of data transfer between global memory and the GPU cores.

  Kernel launch overhead The time taken to launch the kernel on the GPU.

  Occupancy How well the GPU's resources are utilized (number of active warps per SM).

- To measure performance, we can use CUDA events to time the kernel execution and calculate the achieved GFLOPS.
- Optimizations such as using shared memory, minimizing global memory accesses, and ensuring coalesced memory access patterns can significantly improve performance.

CUDA Events for Timing

An approach to measure kernel execution time is to use **CUDA Events**:

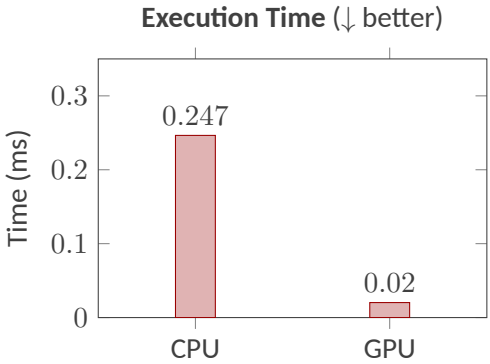
- Create events: `cudaEventCreate`
- Record events: `cudaEventRecord`
- Synchronize: `cudaEventSynchronize`
- Elapsed time: `cudaEventElapsedTime`

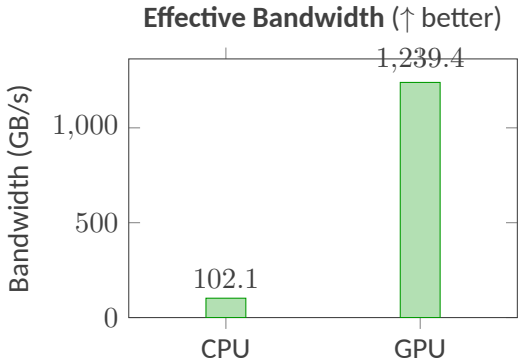
```
cudaEvent_t start, stop;
float ms;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
// Launch kernel
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&ms, start, stop);
```

To calculate GFLOPS:

$$\text{GFLOPS} = \frac{2n}{ms \times 10^6}$$

These are the results on my laptop (RTX 4060)





Speedup: 12.14 x

Vector size: $N = 2^{20}$ elements (8.0 MB)

The cuBLAS Library

- **cuBLAS** is NVIDIA's GPU-accelerated implementation of the Basic Linear Algebra Subprograms (BLAS) library.
- It provides highly optimized routines for dense linear algebra operations, including:
  - Level 1 BLAS: Vector operations (e.g., DAXPY, DOT).
  - Level 2 BLAS: Matrix-vector operations (e.g., GEMV).
  - Level 3 BLAS: Matrix-matrix operations (e.g., GEMM).
- cuBLAS leverages the full capabilities of NVIDIA GPUs, including:
  - Efficient memory access patterns.
  - Use of shared memory and registers.
  - Optimized kernel launches and execution strategies.
- It is widely used in scientific computing, machine learning, and other high-performance applications requiring fast linear algebra computations.

Using cuBLAS

To ensure maximum compatibility with existing Fortran environments, the cuBLAS library operates based on Column-Major storage and 1-based indexing.

Implications for C/C++ Developers:

- C and C++ utilize **Row-Major** storage by default.
- Consequently, native 2D array semantics (e.g., `double A[rows][cols]`) cannot be directly used with cuBLAS.

You can define two macros to help with indexing:

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

A couple of useful macros

Writing **error check code** is very repetitive. We can define a couple of macros to help us:

1) To wrap *CUDA runtime* API calls:

```
#define CHECK_CUDA(call)                                                    \
    do {                                                                    \
        cudaError_t err = call;                                             \
        if (err != cudaSuccess) {                                           \
            fprintf(stderr, "CUDA error %s:%d: %s\n",                       \
                    __FILE__, __LINE__, cudaGetErrorString(err));           \
            exit(EXIT_FAILURE);                                             \
        }                                                                   \
    } while (0)
```

This macro checks the return status of a CUDA runtime API call and prints an error message if the call fails.

A couple of useful macros

Writing **error check code** is very repetitive. We can define a couple of macros to help us:

2) To wrap *cuBLAS runtime* API calls:

```
#define CHECK_CUBLAS(call)                                          \
    do {                                                            \
        cublasStatus_t status = call;                               \
        if (status != CUBLAS_STATUS_SUCCESS) {                      \
            fprintf(stderr, "cuBLAS error %s:%d: %d\n",             \
                    __FILE__, __LINE__, status);                    \
            exit(EXIT_FAILURE);                                     \
        }                                                           \
    } while (0)
```

This macro checks the return status of a cuBLAS API call and prints an error message if the call fails.

cuBLAS example: DAXPY

To use cuBLAS the first thing we need to do is create a **cuBLAS handle**:

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

This handle is used to manage the cuBLAS library context and resources.

Then we can call the `cublasDaxpy` function to perform the DAXPY operation:

```
CHECK_CUBLAS(cublasDaxpy(handle, n, &alpha, d_x, 1, d_y, 1));
```

At this point an **interface** to the DAXPY operation should be **very familiar**!

Finally, we need to destroy the cuBLAS handle to free resources:

```
cublasDestroy(handle);
```

The cuBLAS DAXPY interface


```
cublasStatus_t cublasDaxpy(cublasHandle_t handle, int n,
                           const double          *alpha,
                           const double          *x, int incx,
                           double                *y, int incy);
```

handle cuBLAS library context.

n Number of elements in vectors `x` and `y`.

alpha Pointer to the scalar multiplier.

x Pointer to the input vector $\mathbf{x}$.

incx Stride between elements in `x` (usually 1).

y Pointer to the input/output vector $\mathbf{y}$.

incy Stride between elements in `y` (usually 1).

The cuBLAS DAXPY interface


```
cublasStatus_t cublasDaxpy(cublasHandle_t handle, int n,
                           const double          *alpha,
                           const double          *x, int incx,
                           double                *y, int incy);
```

alpha Pointer to the scalar multiplier.

`alpha` and `beta` parameters can be *passed by reference* on the host or the device.
When the pointer mode is set to `CUBLAS_POINTER_MODE_HOST`:

- The scalars can be on the stack or heap (not managed memory).
- The kernels are launched with the *value* of the scalar.
- Host memory can be freed immediately after the call.

The cuBLAS DAXPY interface


```
cublasStatus_t cublasDaxpy(cublasHandle_t handle, int n,
                           const double          *alpha,
                           const double          *x, int incx,
                           double                *y, int incy);
```

alpha Pointer to the scalar multiplier.

`alpha` and `beta` parameters can be *passed by reference* on the host or the device.
When set to `CUBLAS_POINTER_MODE_DEVICE`:

- The scalars must be accessible on the device.
- Their values must not change until the kernel completes.
- Allows fully asynchronous execution, even if `alpha` is generated by a previous kernel (common in iterative solvers).

The cuBLAS DAXPY interface


```
cublasStatus_t cublasDaxpy(cublasHandle_t handle, int n,
                           const double          *alpha,
                           const double          *x, int incx,
                           double                *y, int incy);
```

alpha Pointer to the scalar multiplier.

`alpha` and `beta` parameters can be *passed by reference* on the host or the device.
To **set the pointer mode**, use:

```
cublasSetPointerMode(handle, CUBLAS_POINTER_MODE_HOST);
```

The *default mode* is `CUBLAS_POINTER_MODE_HOST`.

The cuBLAS 2-norm

The cuBLAS library provides the `cublasDnrm2` function to compute the Euclidean norm (2-norm) of a vector:

```
cublasStatus_t  cublasDnrm2(cublasHandle_t handle, int n,
                            const double *x, int incx, double *result)
```

handle cuBLAS library context.

n Number of elements in vector `x`.

x Pointer to the input vector $\mathbf{x}$.

incx Stride between elements in `x` (usually 1).

result Pointer to store the computed norm.

The cuBLAS 2-norm

The cuBLAS library provides the `cublasDnrm2` function to compute the Euclidean norm (2-norm) of a vector:

```
cublasStatus_t  cublasDnrm2(cublasHandle_t handle, int n,
                            const double *x, int incx, double *result)
```

result Pointer to store the computed norm.

For the functions of this category, when the pointer mode is set to `CUBLAS_POINTER_MODE_HOST`, these functions block the CPU, until the GPU has completed its computation and the results have been copied back to the Host.

The cuBLAS 2-norm

The cuBLAS library provides the `cublasDnrm2` function to compute the Euclidean norm (2-norm) of a vector:

```
cublasStatus_t  cublasDnrm2(cublasHandle_t handle, int n,
                            const double *x, int incx, double *result)
```

result Pointer to store the computed norm.

When the pointer mode is set to `CUBLAS_POINTER_MODE_DEVICE`, these functions return immediately. This requires proper synchronization in order to **read the result from the host**.

The cuBLAS GEMV operation

The cuBLAS library provides the `cublasDgemv` function to perform the matrix-vector multiplication operation:

$$\mathbf{y} \leftarrow \alpha\,\mathrm{op}(\mathbf{A})\mathbf{x} + \beta\mathbf{y}$$

The function signature is:

```
cublasStatus_t cublasDgemv(cublasHandle_t handle,
                           cublasOperation_t trans, int m, int n,
                           const double           *alpha,
                           const double           *A, int lda,
                           const double           *x, int incx,
                           const double           *beta,
                           double *y, int incy)
```

The cuBLAS GEMV operation

handle	cuBLAS library context.
trans	Operation on matrix `A` (`CUBLAS_OP_N`, `CUBLAS_OP_T`, `CUBLAS_OP_C`).
m,n	Dimensions of matrix `A` (rows, columns).
alpha	Pointer to the scalar multiplier for $\mathrm{op}(\mathbf{A})\mathbf{x}$.
A	Pointer to the input matrix $\mathbf{A}$.
lda	Leading dimension of matrix `A` (usually `m`).
x	Pointer to the input vector $\mathbf{x}$.
incx	Stride between elements in `x` (usually 1).
beta	Pointer to the scalar multiplier for $\mathbf{y}$.
y	Pointer to the input/output vector $\mathbf{y}$.
incy	Stride between elements in `y` (usually 1).

We can run a comparison with OpenBLAS

We can create and populate matrices/vectors on the host,

```
const size_t bytes_A = (size_t)m * n * sizeof(double);
const size_t bytes_x = (size_t)n * sizeof(double);
const size_t bytes_y = (size_t)m * sizeof(double);
double *h_A = (double *)malloc(bytes_A);
double *h_x = (double *)malloc(bytes_x);
double *h_y = (double *)malloc(bytes_y);
double *h_y_cpu = (double *)malloc(bytes_y);
double *h_y_gpu = (double *)malloc(bytes_y);
```

We can run a comparison with OpenBLAS

We can create and populate matrices/vectors on the host,

```
for (int col = 0; col < n; col++) {
  for (int row = 0; row < m; row++) {
    h_A[col * m + row] = 1.0 + (row + col) * 1e-6;
  }
}
for (int i = 0; i < n; i++) {
  h_x[i] = 1.0;
}
for (int i = 0; i < m; i++) {
  h_y[i] = 2.0;
  h_y_cpu[i] = 2.0;
}
```

We can run a comparison with OpenBLAS

We can create and populate matrices/vectors on the host, copy them to the device,

```
double *d_A, *d_x, *d_y;
// Allocate device memory
CHECK_CUDA(cudaMalloc(&d_A, bytes_A));
CHECK_CUDA(cudaMalloc(&d_x, bytes_x));
CHECK_CUDA(cudaMalloc(&d_y, bytes_y));
// Copy data to device
CHECK_CUDA(cudaMemcpy(d_A, h_A, bytes_A, cudaMemcpyHostToDevice));
CHECK_CUDA(cudaMemcpy(d_x, h_x, bytes_x, cudaMemcpyHostToDevice));
CHECK_CUDA(cudaMemcpy(d_y, h_y, bytes_y, cudaMemcpyHostToDevice));
```

We can run a comparison with OpenBLAS

We can create and populate matrices/vectors on the host, copy them to the device, and run the `cublasDgemv` function

```
int m = 4096;
int n = 4096;
const double alpha = 2.0;
const double beta = 1.0;
CHECK_CUBLAS(cublasDgemv(handle, CUBLAS_OP_N, m, n, &alpha, d_A,
  m, d_x, 1,
  &beta, d_y, 1));
```

- We run 10 warm-up iterations before timing the execution.
- We then run 100 timed iterations to measure performance.
- Finally, we copy the result back to the host for verification.

We can run a comparison with OpenBLAS

We copy back the result to the host and verify correctness against OpenBLAS:

```
CHECK_CUDA(cudaMemcpy(h_y_gpu, d_y, bytes_y, cudaMemcpyDeviceToHost));
```

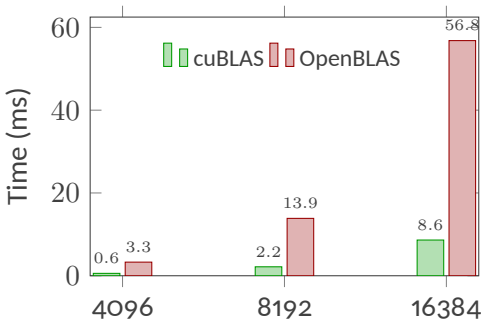
We can run a comparison with OpenBLAS

We copy back the result to the host and verify correctness against OpenBLAS:

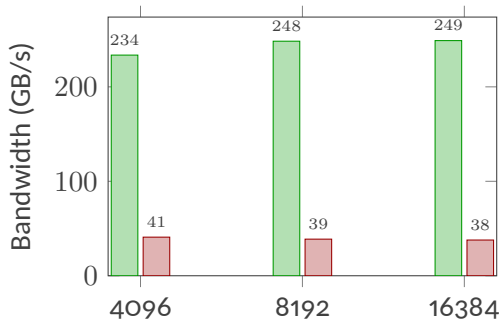
```
CHECK_CUDA(cudaMemcpy(h_y_gpu, d_y, bytes_y, cudaMemcpyDeviceToHost));
cblas_dgemv(CblasColMajor, CblasNoTrans,m, n, alpha, h_A, m, h_x, 1, beta,
↪  h_y_cpu, 1);
// Verify correctness
int errors = 0;
for (int i = 0; i < m && errors < 10; i++) {
  double diff = fabs(h_y_cpu[i] - h_y_gpu[i]);
  if (diff > 1e-8) {
    fprintf(stderr, "Mismatch at %d: CPU %.12f GPU %.12f\n",
        i, h_y_cpu[i], h_y_gpu[i]);
    errors++;
  }
}
```

Comparison with OpenBLAS (NVIDIA RTX 4060 GPU)

Execution Time (↓ better)



Effective Bandwidth (↑ better)



Speedup of roughly **6.5x**, saturating the GPU memory bandwidth at $\sim 249\,\mathrm{GB\,s^{-1}}$.

The cuBLAS GEMM operation

The cuBLAS library provides the `cublasDgemm` function to perform the matrix-matrix multiplication operation:

$$\mathbf{C} \leftarrow \alpha\,\mathrm{op}(\mathbf{A})\,\mathrm{op}(\mathbf{B}) + \beta\mathbf{C}$$

The function signature is:

```
cublasStatus_t cublasDgemm(cublasHandle_t handle,
                           cublasOperation_t transa,
                           cublasOperation_t transb,
                           int m, int n, int k,
                           const double          *alpha,
                           const double          *A, int lda,
                           const double          *B, int ldb,
                           const double          *beta,
                           double          *C, int ldc)
```

The cuBLAS GEMM operation

handle	cuBLAS library context.
transa, transb	Operations on matrices `A` and `B` (`CUBLAS_OP_N`, `CUBLAS_OP_T`, `CUBLAS_OP_C`).
m,n,k	Dimensions of the matrices.
alpha	Pointer to the scalar multiplier for $\mathrm{op}(\mathbf{A})\ \mathrm{op}(\mathbf{B})$.
A	Pointer to the input matrix $\mathbf{A}$.
lda	Leading dimension of matrix `A` (usually `m`).
B	Pointer to the input matrix $\mathbf{B}$.
ldb	Leading dimension of matrix `B` (usually `k`).
beta	Pointer to the scalar multiplier for $\mathbf{C}$.
C	Pointer to the input/output matrix $\mathbf{C}$.
ldc	Leading dimension of matrix `C` (usually `m`).

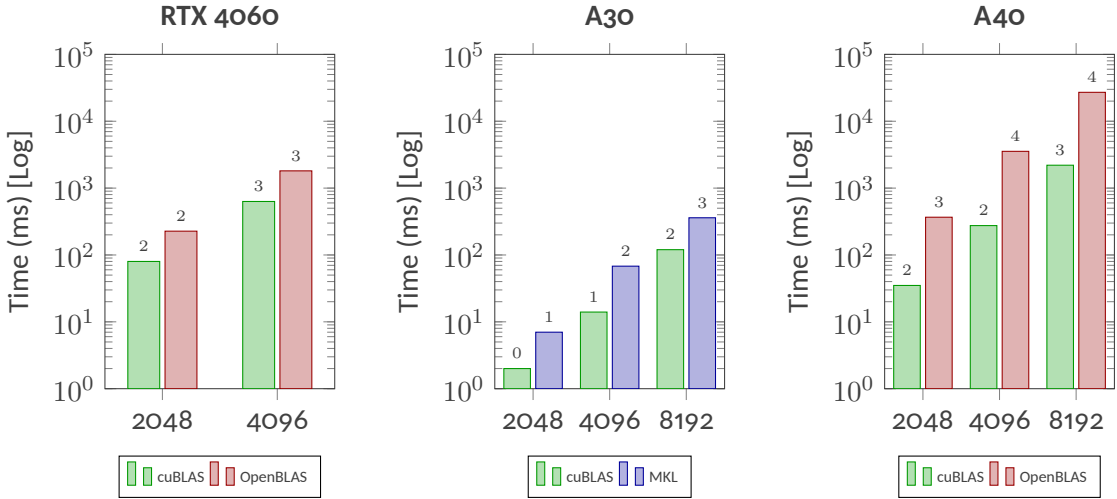
Running a comparison with OpenBLAS

We can run a **similar comparison** as before, but this time using the `cublasDgemm` function to perform matrix-matrix multiplication.

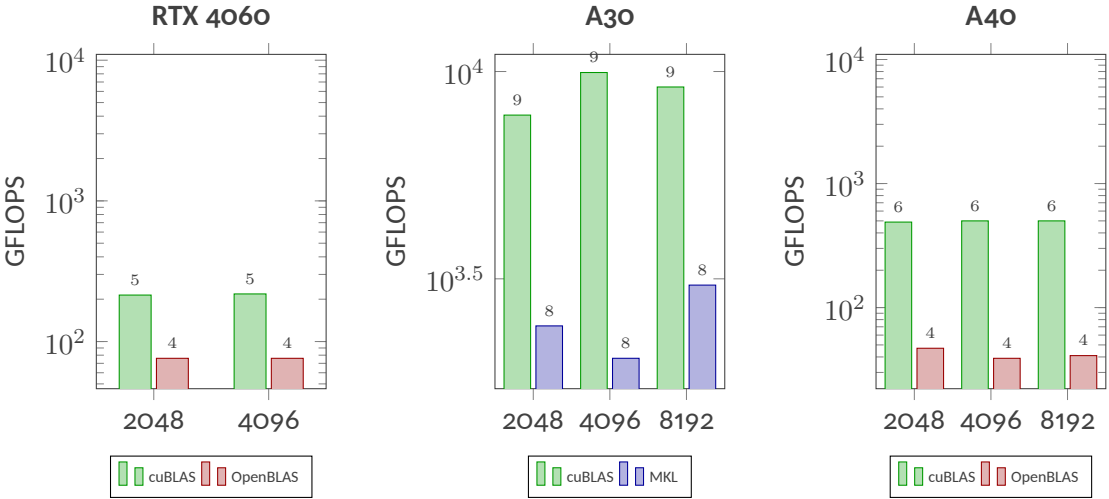
The process involves:

1. Creating and populating matrices on the host.
2. Copying them to the device.
3. Running the `cublasDgemm` function.
4. Copying back the result to the host.
5. Verifying correctness against OpenBLAS.

DGEMM Performance: Throughput



DGEMM Performance: Throughput



Interpreting DGEMM Performance Results

- **A30: Near-Peak FP64 Performance**
  - Based on GA100 architecture with **FP64 Tensor Cores**.
  - cuBLAS DGEMM exploits Tensor Cores when matrix sizes and alignment allow.
  - Measured performance ($\sim$8–10 TFLOPS) reaches **80–90% of theoretical FP64 Tensor Core peak**.
  - Low memory bandwidth utilization confirms the kernel is **compute-bound**.
- **A40: Limited by Scalar FP64 Units**
  - Based on GA102 architecture with **no FP64 Tensor Cores**.
  - FP64 throughput is limited to **1/64 of FP32 rate**.
  - Observed $\sim$0.5 TFLOPS is consistent with the $\sim$1.1 TFLOPS hardware peak.
  - DGEMM performance is fundamentally constrained by narrow FP64 pipelines.
- **Key Takeaway**
  - Large FP64 performance gap reflects **architectural design choices**.
  - A30 is optimized for **HPC and numerical linear algebra**.

cuSOLVER for LAPACK-like Workloads

- **cuSolverDN** is designed to solve dense linear systems of the form:

$$Ax = b$$

  where $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$.

- **Factorizations provided:**
  - **LU** with partial pivoting for general matrices.
  - **QR** factorization.
  - **Cholesky** for symmetric/Hermitian positive definite matrices.
  - **LDL** (Bunch-Kaufman) for symmetric indefinite matrices.
- **Decompositions:**
  - Singular Value Decomposition (SVD).
  - Bidiagonalization.

cuSOLVER for LAPACK-like Workloads

- **cuSolverDN** is designed to solve dense linear systems of the form:

$$Ax = b$$

  where $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$.

- **Design Philosophy:**
  - Targets computationally intensive LAPACK routines.
  - Provides an API compatible with LAPACK.
  - The main idea is to allows users to accelerate bottlenecks on the GPU while keeping other parts of the code on the CPU.

Solving a Linear System with cuSOLVERDN

To solve a linear system $Ax = b$ using the cuSOLVERDN library, the process typically involves two main steps, mirroring the LAPACK approach:

1. **Factorization:** Compute the LU factorization of the coefficient matrix A using `cusolverDnDgetrf`. This decomposes A into $P \cdot L \cdot U$.
2. **Solve:** Solve the system using the computed factors with `cusolverDnDgetrs`. This involves forward and backward substitutions.

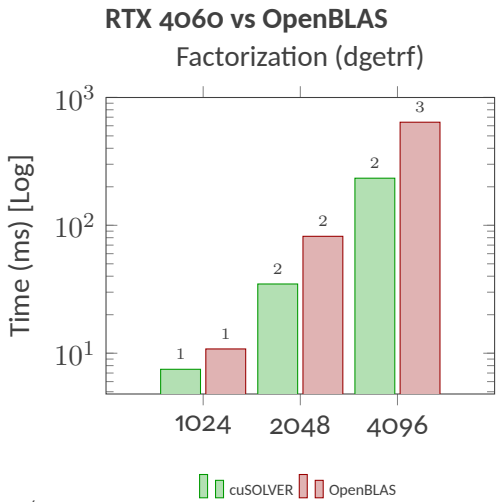
Solving a Linear System with cuSOLVERDN

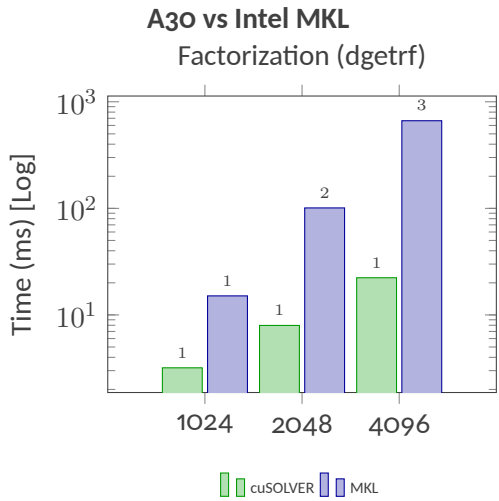
To solve a linear system $Ax = b$ using the cuSOLVERDN library, the process typically involves two main steps, mirroring the LAPACK approach.

Main Bottlenecks on GPUs:

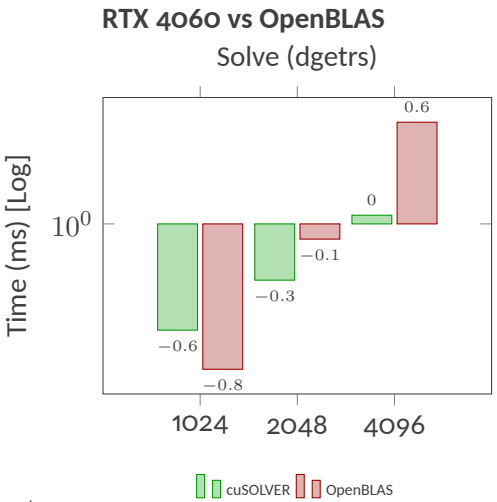
- **Pivoting:** The LU factorization requires partial pivoting for numerical stability. This involves finding the pivot element (reduction) and swapping rows (irregular memory access), which are sequential and memory-bound operations that hurt GPU parallelism.
- **Triangular Solves (TRSM):** The **forward** and **backward substitutions** in `getrs` have dependencies between rows/columns, limiting the available parallelism compared to matrix multiplication (GEMM).

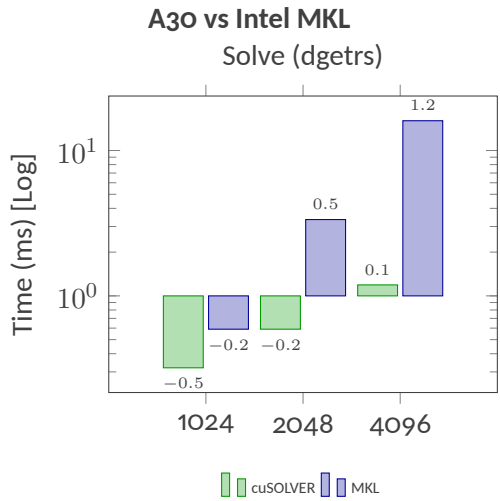
cuSOLVERDN Performance Factorization Throughput



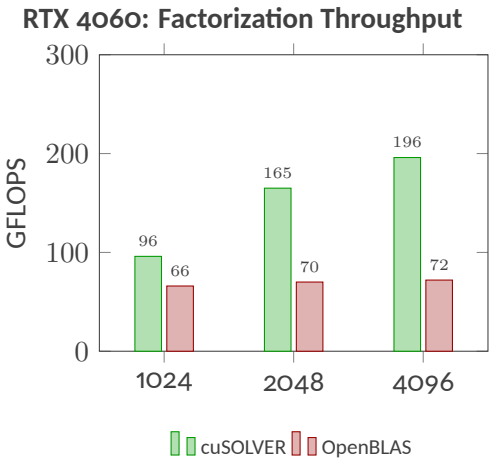


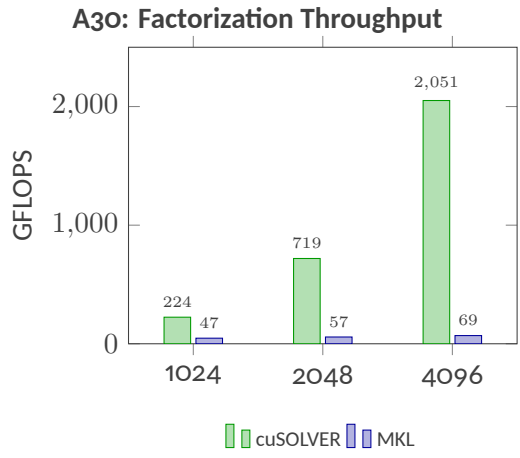
cuSOLVERDN Performance Factorization Throughput





cuSOLVERDN Performance Factorization Throughput





Interpreting cuSOLVERDN Performance Results

- **Factorization (dgetrf):**
  - cuSOLVERDN achieves significant speedups over CPU libraries (OpenBLAS, MKL) for LU factorization.
  - The GPU's parallelism effectively accelerates the computationally intensive parts of the factorization.
  - However, performance is still limited by pivoting and memory-bound operations.
- **Solve (dgetrs):**
  - The solve step shows more modest speedups due to inherent sequential dependencies in triangular solves.
  - While GPUs can accelerate some parts, the limited parallelism restricts overall performance gains.
- **Overall Takeaway:**
  - cuSOLVERDN provides performance improvements for dense linear algebra tasks on GPUs.
  - The effectiveness varies between factorization and solve phases.

The other cuSOLVERDN Routines

The cuSOLVERDN library also provides other LAPACK-like routines, each with its own performance characteristics on GPUs:

- **QR Factorization (dgeqrf)**: Similar performance characteristics to LU factorization, with speedups over CPU libraries.
- **Cholesky Factorization (dpotrf)**: Generally faster than LU due to the absence of pivoting, achieving higher throughput on GPUs.
- **SVD (dgesvd)**: More complex and computationally intensive, with performance gains depending on matrix size and GPU capabilities.
- **Bidiagonalization (dgebd2)**: Similar to SVD, with performance influenced by the algorithmic complexity and data movement.

The other cuSOLVERDN Routines

The cuSOLVERDN library also provides other LAPACK-like routines, each with its own performance characteristics on GPUs:

- **QR Factorization (dgeqrf)**
- **Cholesky Factorization (dpotrf)**
- **SVD (dgesvd)**
- **Bidiagonalization (dgebd2)**

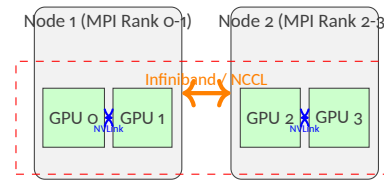
As we have said many times, it is crucial to:

- Profile your specific workload.
- Understand the performance characteristics of each routine on your target GPU!
- Test always on your Workloads!

cuSOLVERMp: Distributed Dense Linear Algebra

cuSOLVERMp is the NVIDIA library for distributed-memory dense linear algebra, designed to scale across multiple GPUs and nodes.

- **Multi-process, Multi-GPU**:
  - Follows the *One process per GPU* paradigm.
  - Seamless integration with MPI applications.
- **ScaLAPACK Compatibility**:
  - C interfaces designed to **mirror ScaLAPACK**.
  - Porting of legacy distributed CPU codes.
- **High Performance**:
  - Uses **NCCL** (NVIDIA Collective Communication Library) for inter-GPU communication.
  - **Tensor Core** accelerated math kernels.
- **Tools**: Built-in logging and tracing support.



cuSOLVERMp Logic

cuSOLVERMp: Workflow and Data Layout

Data Layout of Local Matrices

- cuSOLVERMp assumes that local matrices are stored in column-major format (compatible with Fortran/LAPACK).

Workflow Overview

1. Create a NCCL communicator (NCCL Initialization).
2. Initialize the library handle: `cusolverMpCreate()`.
3. Initialize grid descriptors: `cusolverMpCreateDeviceGrid()`.
4. Initialize matrix descriptors: `cusolverMpCreateMatrixDesc()`.
5. Query the host and device buffer sizes for a given routine.
6. Allocate host and device workspace buffers.
7. Execute the routine to perform the desired computation.
8. Synchronize local stream: `cudaStreamSynchronize()`.
9. Cleanup resources (workspaces, descriptors, handle, communicator).

cuSOLVERMp error control macro

Like for the other CUDA libraries, it is a good idea to define a macro for error checking:

```
#define CHECK_CUSOLVER_MP(call)                                    \
    do {                                                           \
        cusolverStatus_t status = call;                            \
        if (status != CUSOLVER_STATUS_SUCCESS) {                   \
            fprintf(stderr, "cusolverMp error %s:%d: %d\n",        \
                    __FILE__, __LINE__, status);                   \
            exit(EXIT_FAILURE);                                    \
        }                                                          \
    } while (0)
```

This works exactly like the error checking macros we have seen for cuBLAS and cuSOLVERDN, but adapted for the cuSOLVERMp API.

cuSOLVERMp Initialization: Overview

The initialization process for cuSOLVERMp closely mirrors the distributed memory setup we saw with ScaLAPACK/BLACS, but with the addition of GPU-specific communication layers (NCCL).

ScaLAPACK / BLACS

1. Initialize MPI.
2. Initialize BLACS (Process Grid).
3. Create Context / descriptors.

cuSOLVERMp

1. Initialize MPI.
2. Set CUDA device context.
3. Initialize NCCL (GPU Comms).
4. Create library handle & Grids.

Key Data Types:

```
ncclUniqueId id; // Unique identifier for NCCL comm
ncclComm_t comm; // NCCL communicator handle
```

Step 1 & 2: MPI and Device Setup

Step 1: MPI Setup

Standard MPI initialization, just like any distributed application.

```
MPI_Init(nullptr, nullptr);
int rank, nranks;
MPI_Comm_size(MPI_COMM_WORLD, &nranks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Step 2: CUDA Device Setup

Assign a specific GPU to the MPI process (usually rank i gets GPU $i \pmod{n}_{gpus}$).

```
const int local_device = getLocalDevice(); // e.g., rank % num_gpus
CUDA_CHECK(cudaSetDevice(local_device));
CUDA_CHECK(cudaFree(nullptr));  // Initialize CUDA context
```

Step 3: NCCL Communicator Creation

This step is roughly equivalent to `Cblacs_gridinit`, but for the GPU interconnect.

```
ncclUniqueId id;
// Rank 0 generates the unique ID
if (rank == 0) {
    NCCL_CHECK(ncclGetUniqueId(&id));
}
// Broadcast the ID to all ranks so they can join the same communicator
MPI_CHECK(MPI_Bcast(&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD));
// Initialize NCCL communicator
ncclComm_t comm;
NCCL_CHECK(ncclCommInitRank(&comm, nranks, id, rank));
```

Step 4, 5 & 6: Handles and Grids

Step 4: Stream Creation

```
cudaStream_t stream = nullptr;
CUDA_CHECK(cudaStreamCreate(&stream));
```

Step 5: Library Handle

```
cusolverMpHandle_t handle = nullptr;
CUSOLVER_CHECK(cusolverMpCreate(&handle, local_device, stream));
```

Step 6: Process Grid (Equivalent to BLACS Grid)

```
cusolverMpGrid_t grid = nullptr;
// Map columns of the process grid to columns of the matrix
CUSOLVER_CHECK(cusolverMpCreateDeviceGrid(handle, &grid, comm,
                nprow, npcol, CUSOLVERMP_GRID_MAPPING_COL_MAJOR));
```

Cleanup and Synchronization

Synchronization

Since cuSOLVERMp is asynchronous, sync the stream before checking results or exiting.

```
CUDA_CHECK(cudaStreamSynchronize(stream));
```

Cleanup: *after we have done everything*

Proper destruction order is crucial (reverse of initialization).

```
// Destroy cuSOLVERMp objects first
CUSOLVER_CHECK(cusolverMpDestroyGrid(grid));
CUSOLVER_CHECK(cusolverMpDestroy(handle));
// Destroy NCCL communicator
NCCL_CHECK(ncclCommDestroy(comm));
// Clean up CUDA resources
CUDA_CHECK(cudaStreamDestroy(stream));
// Finalize MPI
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
```

Matrix Management: Descriptors

Just like in ScaLAPACK (where we use `descinit`), cuSOLVERMp requires **matrix descriptors** to understand the **data distribution** (2D block-cyclic).

```
cusolverStatus_t cusolverMpCreateMatrixDesc(
    cusolverMpMatrixDescriptor_t *descr,
    cusolverMpGrid_t grid,
    cudaDataType dataType, // CUDA_R_64F for double
    int64_t M_A, int64_t N_A,            // Global Dimensions
    int64_t MB_A, int64_t NB_A,          // Block sizes
    uint32_t RSRC_A, uint32_t CSRC_A,    // Origin (usually 0,0)
    int64_t LLD_A);                      // Local Leading Dimension
```

- **Parallel with ScaLAPACK**: This is the direct equivalent of the array descriptor array `DESC_` (e.g., `DESC_A`).
- Defines how the global matrix is mapped to the process grid.

Utility: Calculating Local Sizes

Before **allocating device memory**, we need to know how much memory the local portion of the distributed matrix requires.
cuSOLVERMp provides a helper equivalent to ScaLAPACK's `NUMROC`:

```
int64_t cusolverMpNUMROC(
    int64_t n,          // Global dimension (rows or cols)
    int64_t nb,         // Block size
    uint32_t iproc,     // My coordinate (row or col)
    uint32_t isrcproc,  // Source coordinate (usually 0)
    uint32_t nprocs);   // Total processes in dimension
```

Utility: Calculating Local Sizes

Before **allocating device memory**, we need to know how much memory the local portion of the distributed matrix requires.
cuSOLVERMp provides a helper equivalent to ScaLAPACK's `NUMROC`:

```
int64_t m_local = cusolverMpNUMROC(M, MB, proc_row, 0, nprow);
int64_t n_local = cusolverMpNUMROC(N, NB, proc_col, 0, npcol);
// Allocate on GPU
CUDA_CHECK(cudaMalloc(&d_A, m_local * n_local * sizeof(double)));
```

Data Distribution Utilities

To simplify testing and porting, cuSOLVERMp provides utilities to scatter/gather data between a single host process and the distributed device memory.

`cusolverMpMatrixScatterH2D`

- Scatters a global matrix (on Host) to distributed matrices (on Device).
- Only for testing/debugging (not high performance).

```
cusolverMpMatrixScatterH2D(handle, M, N, d_A, IA, JA, descrA,
                           root, h_src, h_ldsrc);
```

`cusolverMpMatrixGatherD2H`

- Gathers a distributed matrix (from Device) to a global matrix (on Host).
- Useful for verifying results.

Data Distribution Utilities

To simplify testing and porting, cuSOLVERMp provides utilities to scatter/gather data between a single host process and the distributed device memory.

`cusolverMpMatrixScatterH2D`

- Scatters a global matrix (on Host) to distributed matrices (on Device).
- Only for testing/debugging (not high performance).

`cusolverMpMatrixGatherD2H`

- Gathers a distributed matrix (from Device) to a global matrix (on Host).
- Useful for verifying results.

```
cusolverMpMatrixGatherD2H(handle, M, N, d_A, IA, JA, descrA,
                          root, h_dst, h_lddst);
```

Available Dense API

The available dense API are:

- `cusolverMpGetrf` which **computes the LU factorization** of a general matrix.
  - `cusolverMpGetrf_bufferSize` which computes the size of the workspace needed for `cusolverMpGetrf`.
- `cusolverMpGetrs` which **solves a system of linear equations** with a general matrix using the LU factorization computed by `cusolverMpGetrf`.
  - `cusolverMpGetrs_bufferSize` which computes the size of the workspace needed for `cusolverMpGetrs`.
- `cusolverMpPotrf` which computes the Cholesky factorization of a symmetric positive definite matrix.
  - `cusolverMpPotrf_bufferSize` which computes the size of the workspace needed for `cusolverMpPotrf`.

Available Dense API

- `cusolverMpPotrs` which **solves a system of linear equations** with a **symmetric positive definite** matrix using the Cholesky factorization computed by `cusolverMpPotrf`.
  - `cusolverMpPotrs_bufferSize` which computes the size of the workspace needed for `cusolverMpPotrs`.
- `cusolverMpGeqrf` which **computes the QR factorization** of a general matrix.
  - `cusolverMpGeqrf_bufferSize` which computes the size of the workspace needed for `cusolverMpGeqrf`.
- `cusolverMpOrmqr` which multiplies a matrix by the orthogonal matrix Q from a QR factorization computed by `cusolverMpGeqrf`.
  - `cusolverMpOrmqr_bufferSize` which computes the size of the workspace needed for `cusolverMpOrmqr`.

Available Dense API

- `cusolverMpGels` which **solves a system of linear equations** with a general matrix using the QR factorization computed by `cusolverMpGeqrf`.
  - `cusolverMpGels_bufferSize` which computes the size of the workspace needed for `cusolverMpGels`.
- `cusolverMpSytrd` which reduces a symmetric matrix to tridiagonal form (Schur Decomposition).
  - `cusolverMpSytrd_bufferSize` which computes the size of the workspace needed for `cusolverMpSytrd`.
- `cusolverMpStedc` which computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.
  - `cusolverMpStedc_bufferSize` which computes the size of the workspace needed for `cusolverMpStedc`.

Available Dense API

- `cusolverMpOrmtr` which multiplies a matrix by the orthogonal matrix Q from a symmetric tridiagonal reduction computed by `cusolverMpSytrd`.
  - `cusolverMpOrmtr_bufferSize` which computes the size of the workspace needed for `cusolverMpOrmtr`.
- `cusolverMpSyevd` which computes all eigenvalues and, optionally, eigenvectors of a symmetric matrix using the divide and conquer method.
  - `cusolverMpSyevd_bufferSize` which computes the size of the workspace needed for `cusolverMpSyevd`.
- `cusolverMpSygst` which reduces a symmetric-definite generalized eigenvalue problem to standard form.
  - `cusolverMpSygst_bufferSize` which computes the size of the workspace needed for `cusolverMpSygst`.

Available Dense API

- `cusolverMpSygvd` which computes all eigenvalues and, optionally, eigenvectors of a symmetric-definite generalized eigenvalue problem.
  - `cusolverMpSygvd_bufferSize` which computes the size of the workspace needed for `cusolverMpSygvd`.

⚠ Each of these routines has a corresponding `_bufferSize` function that allows you to query the amount of workspace memory needed before performing the actual computation.

This is **crucial for efficient memory management** in distributed GPU environments.

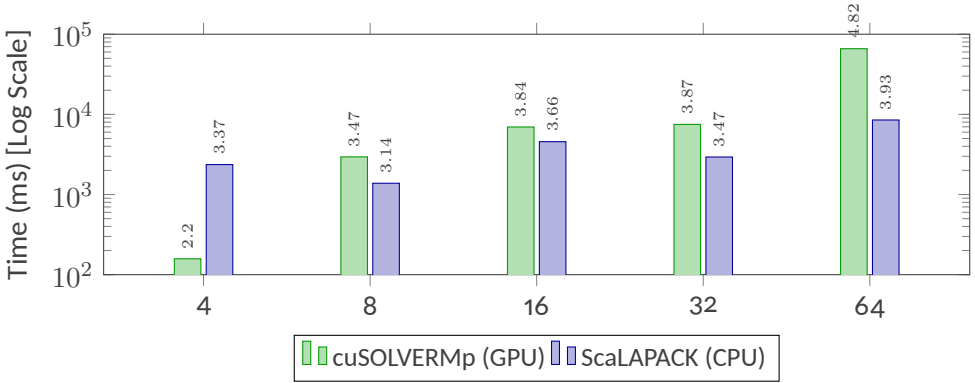
Weak Scaling (Large Matrix): Throughput

We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.

- We use the routine `cusolverMpGetrf` for cuSOLVERMp and `PDGETRF` for ScaLAPACK.
- Configuration for nodes 1 to 16:
  - 4 Tasks per Node, 1 GPU per Task (on NVIDIA A30 GPUs)
  - 4 Tasks per Node, 16 CPU cores per Task (Intel® Xeon® Gold 6338 CPU @ 2.00GHz)
- Matrix Size per Process: $N_{local} = 4096$
- Matrix Size per Process: $N_{local} = 16384$

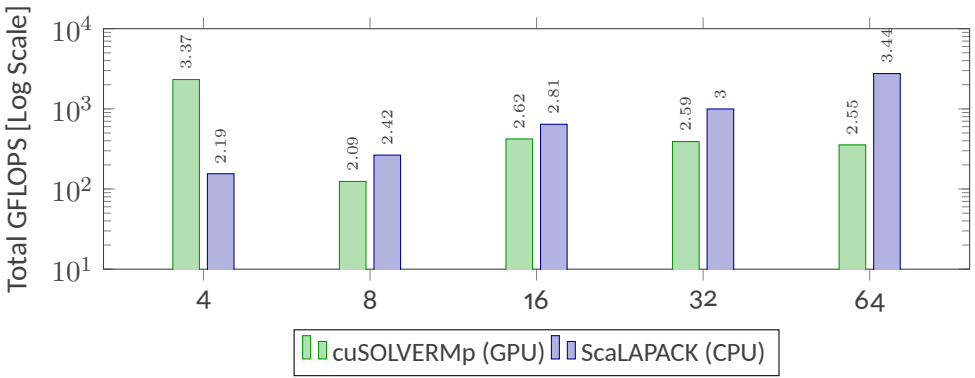
Weak Scaling (Large Matrix): Throughput

We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.



Weak Scaling (Large Matrix): Throughput

We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.



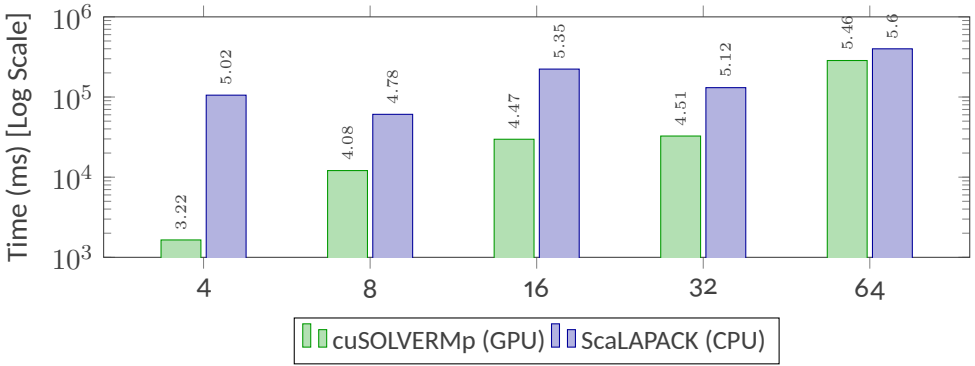
Weak Scaling (Large Matrix): Throughput

We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.

- We use the routine `cusolverMpGetrf` for cuSOLVERMp and `PDGETRF` for ScaLAPACK.
- Configuration for nodes 1 to 16:
  - 4 Tasks per Node, 1 GPU per Task (on NVIDIA A30 GPUs)
  - 4 Tasks per Node, 16 CPU cores per Task (Intel® Xeon® Gold 6338 CPU @ 2.00GHz)
- Matrix Size per Process: $N_{local} = 4096$
- Matrix Size per Process: $N_{local} = 16384$

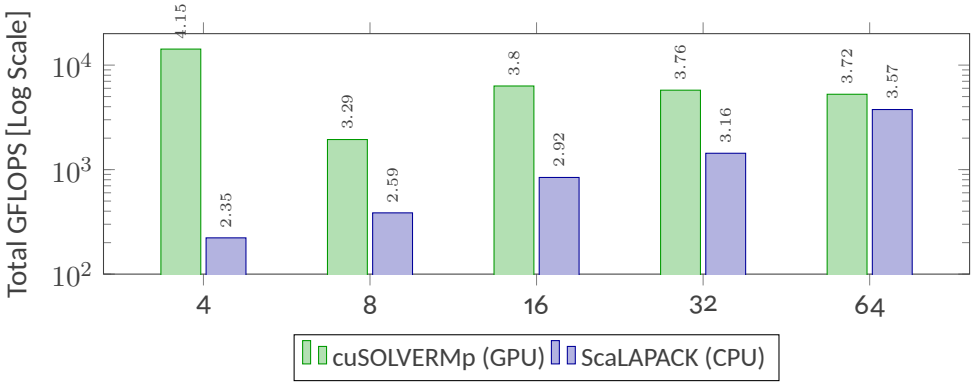
Weak Scaling (Large Matrix): Throughput

We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.



Weak Scaling (Large Matrix): Throughput

We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.



Interpreting the Distributed LU Results

The comparison emphasizes the different sweet spots for CPU and GPU clusters:

- **Massive Throughput on Large Problems**: For $N_{local} = 16384$, cuSOLVERMp is **orders of magnitude faster**. The A30 GPUs exploit Tensor Cores to deliver over 14 P on small rank counts, crushing the CPU performance. This confirms that for *large-scale dense problems*, GPUs are the superior choice.
- **The Latency penalty on Small Problems**: For $N_{local} = 4096$ distributed across many ranks (e.g., 64), the GPU performance collapses.
  - Computation is too fast to hide communication latency (pivoting, panel exchange).
  - ScaLAPACK on CPUs scales better here because the **slower CPU compute allows for better overlapping with communication**, and CPUs generally handle fine-grained dependencies (like pivoting) with less latency overhead relative to their compute speed.
- **Efficiency**: Weak scaling on GPUs requires **keeping the problem size large** to maintain high efficiency. If the local matrix shrinks or stays constant but relatively small (< 10k), the interconnect (PCIe/NVLink) becomes the bottleneck.

Wrapping up the course

3 Conclusions

In this course, we have covered:

- ✔ Some fundamentals of **p**arallel **c**omputing on modern **HPC** systems
  - A quick overview of computer architectures,
  - The difference between Shared Memory and Distributed Memory systems,
  - The main programming models for each of these architectures (**OpenMP** and **MPI**),
  - Programming in modern Fortran.
- ✔ The fundamentals of **BLAS libraries**:
  - in the *Shared Memory* Context via OpenMP and CPU vectorization,
  - in the *Distributed Memory* Context via MPI,
  - in the *GPU* Context via **CUDA**.
- ✔ The fundamentals of **NLA algorithms** and *libraries*:
  - The LAPACK library for Shared Memory systems,
  - The ScaLAPACK library for Distributed Memory systems,
  - The cuSOLVER library for GPU-accelerated systems.

What remains to do?

3 Conclusions

We have covered a lot of ground, but there is still much more to explore in the world of HPC and numerical linear algebra:

- There is the world of **Sparse Linear Algebra** that we have not touched upon.
- There are many more **advanced algorithms** and **libraries** to explore (e.g., **MAGMA**, **PLASMA**, **PSCToolkit**, **PETSc**, **Trilinos**, etc.).
- Performance tuning and optimization for specific architectures is a deep field in itself.
- Emerging architectures (e.g., **TPUs**, **FPGAs**) and programming models (e.g., **SYCL**, **Kokkos**) are worth exploring.

The way forward

On a **shorter term**, look for a problem that interests you, and try to implement and optimize a solution using the tools and techniques we have discussed in this course!