Summer Student Program Report

Project Title:

Device side linear algebra libraries / Cooperative groups implementation

Student Name: František Stloukal

Start Date: 1/7/2024 End Date: 6/9/2024

Name of University:

Czech technical university in Prague

CERN Supervisor:

Dr. Andrea Bocci

1 Report on the GPU libraries

Introduction

The aim of this project was to implement GPU device-side libraries for linear algebra operations in the context of the Pixeltrack-Standalone framework, specifically for the code of PixelTriplets - Broken line. The focus was on evaluating the performance and suitability of MAGMA and cuBLASDx libraries for this purpose. The core requirement involved performing small-scale linear algebra operations, particularly the Cholesky decomposition, used for factorizing positive-definite matrices into the product of a lower triangular matrix and its transpose (LL^T), on matrices as small as 6×6.

MAGMA Library Evaluation

The MAGMA library was initially considered based on a recommendation from a colleague of my supervisor. However, the absence of proper documentation posed significant challenges. Despite attempts to contact the developers for assistance, no response was received. This lack of support and documentation led us to abandon further exploration of MAGMA for this task.

cuBLASDx Library Evaluation

The cuBLASDx library was then selected for evaluation. This library supports device-side execution of matrix-matrix multiplication (GEMM). However, as the scope of operations in cuBLASDx is currently limited, and support for key operations such as the Cholesky decomposition (required for our use case) is lacking for version 0.1.0, library should be examined in future.

One critical limitation observed with cuBLASDx was its requirement for a minimum of 32 threads per block (equivalent to a single warp). To achieve any meaningful performance gains problems need to be sufficiently large. Given that our problem domain involved small matrices (maximum 6×6), the overhead associated with using 32 threads was substantial. As a result, this approach did not outperform the one-thread-per-fit method using the EIGEN library, which is optimized for such small-scale problems.

Given previously stated facts proposed approach was followed.

Algorithm 1 GEMM with cuBLASDx

1: **Launch Kernel**
2: Create array of matrices A, B, C in static shared memory.
3: ... do some work ...
4: **for** $i \leftarrow 0$ **to** $BlockSize$ **do**
5: $\quad GEMM(A[i] \times B[i] = C[i])$
6: **end for**
7: ... do some more work ...
8: **End Kernel**

In step 2 array of EIGEN matrices needed to be created in static shared memory for each problem size (3,4,5,6). The size of the array was equal to number of threads in the block. As step 3 and 7 were usually composed of *device* functions which expected an array on which single thread was operating, when GEMM was performed jumping out of that *device* function was necessary. After in steps 4 to 6 all 32 threads were stalled and operated on GEMM for single matrix in the array. After rest of the work could have been done.

As examined code involved several coordinate system transformations (both sides jacobian mult. of matrix of the system), usually two GEMM were performed in success. As many GEMM were present in the code main *device* functions were divided in many more, which led to not well readable code.

Implementation Details

Transfer of the matrices to shared memory needs to be done by user and is required for any operation by cuBLASDx. No support is provided by cuBLASDx for transfering of the matrices.

No special error cathing is provided.

During implementation were encountered differences between cuBLAS and cuBLASDx in terms of storage order. While cuBLAS uses column-major order (as is standard in Fortran), cuBLASDx operates with row-major order, which aligns with C/C++ conventions. This difference necessitated careful management of memory layouts when transferring data between different contexts. This fact was not mentioned anywhere in documentation, when possibility of usage was examined.

Another advantage of cuBLASDx was the simplicity of its interface: there is no need to create or destroy instances of solvers, unlike traditional libraries that require initialization and cleanup steps. However, it is important to note that cuBLASDx requires a GPU with compute capability 7.0 or higher, which corresponds to the `Volta` architecture and above.

Conclusion

Our evaluation of GPU device-side libraries for linear algebra in Pixeltrack-Standalone revealed significant challenges in using MAGMA due to a lack of documentation and support. While cuBLASDx offers a promising interface and efficient execution for certain operations, its limited functionality and inefficiency in handling small matrices make it less suitable for our specific requirements. Consequently, for the small-scale linear algebra tasks encountered in our project, the one-thread-per-fit approach using EIGEN remains the preferred method as it was observed to be at least ***4x faster***.

Example

Example of 2×2 double precision, real numbers, not transposed matrices, $C = \alpha op(A)op(B) + \beta C$ on 860 architecture with smallest possible block size (32 = warp) cuBLASDx GEMM definition is given. Example anticipates A,B and C to be EIGEN matrices already reside in shared memory.

```
#include <cublasdx.hpp>



 using GEMM =decltype(  cublasdx::Size<2,2,2>()+
                        cublasdx::Precision<double>()+
                        cublasdx::Type<cublasdx::type::real>()+
                        cublasdx::TransposeMode<
                        cublasdx::transpose_mode::non_transposed,
                        cublasdx::transpose_mode::non_transposed>()+
                        cublasdx::Function<cublasdx::function::MM>()+
                        cublasdx::SM<860>()+
                        cublasdx::LeadingDimension<2,2,2>()+
                        cublasdx::Block()+
                        cublasdx::BlockDim<32>());

double alpha = 1.0f;
double beta = 0.0f;

GEMM().execute(alpha, A.data(), B.data(), beta, C.data());
```

2 Report on the Cooperative groups

Introduction

CUDA Cooperative Groups represent a approach to managing parallelism within GPU kernels by allowing finer control over thread synchronization within groups of threads. Unlike the previous one-thread-per-fit model, Cooperative Groups enable the use of multiple threads per fit, facilitating parallelization of for-loops.

I explored the application of CUDA Cooperative Groups, specifically Block Tiles, within two distinct contexts: the PixelTracker standalone—PixelTriplets Broken Line algorithm and the CMSSW ECAL Minimization function (FNNLS). I examined the potential of Cooperative Groups to parallelize key sections of these algorithms, optimizing the overall performance.

CUDA Cooperative Groups

CUDA Cooperative Groups provide a flexible mechanism for partitioning threads into subgroups (Block Tiles) and synchronizing them with a finer granularity than what is possible with traditional CUDA synchronization primitives as `__syncthreads()`. Cooperative Groups allow for more natural expression of parallel algorithms and make it easier to ensure correct synchronization.

Block Tiles are particularly useful as they allow for dividing a thread block into smaller groups of threads that can cooperate on shared tasks. These groups must be of sizes 2^n (e.g., 2, 4, 8, 16, 32) to ensure efficient memory and computational usage. When using Cooperative Groups, all data which to be worked on must reside in shared memory.

Application to PixelTracker Standalone — PixelTriplets Broken Line

Algorithm Overview

The PixelTriplets Broken Line algorithm in the PixelTracker standalone framework involves fitting a helix to the trajectory of a charged particle in the detector. This is achieved by combining a circular fit in the transverse plane with a linear fit in the longitudinal plane. The fit is performed by solving a system of equations involving a covariance matrix, with the algorithm divided into several stages: data preparation, fast fit, line fit, and circle fit.

Memory Considerations

To leverage Cooperative Groups in this algorithm, the amount of shared memory required was carefully calculated. The memory needed for each fit was found to be small enough to use static shared memory. The algorithm is templated for handling 3, 4, or 5 hits, with the largest data structure being 6×6. Each data structure must be replicated K times, where K is the number of groups (Block Tiles) into which the thread block is divided.

Arrays of EIGEN matrices can be directly initialized in static shared memory in compile time, although this generates warnings from `nvcc`.

Three groups of variables were identified for placement in shared memory: (1) variables that are purely inputs, (2) those needed within parallelized for-loops, and (3) those used for linear algebra operations. Even in the worst case—handling 5 hits with all three groups in shared memory—17 groups could be supported per block using only static shared memory.

Cooperative groups implementation

Subsequent experiments determined that the optimal block size for this algorithm was 32 threads. This configuration allowed for the introduction of thread groups as small as 2 threads, although groups of 4 were

performing the best, enabling the parallelization of for-loops using Cooperative Groups as described below briefly.

Single thread for loop:

```
for(int i =0; i < N; i++){
    // ... do some work here ..
}
```

Cooperative groups approach in scope of for cycles:

```
#include <cooperative_groups.h>

    //cooperative group magic
namespace cg = cooperative_groups;
cg::thread_block block = cg::this_thread_block();
cg::thread_block_tile<NumberOfThreadsPerTile> tile
    = cg::tiled_partition<NumberOfThreadsPerTile>(block);

for(int i =tile.thread_rank(); i < N; i+= numThreads){
    // ... do some work here ..
}
tile.sync();

    //or if the for cycle goes to the exactly group size
int i = tile.thread_rank();
    // ... do some work here for [i]..
tile.sync();
```

2.1 Performance examination - Local

Several different approaches were examined and the kernel was profiled with `NSight System` on my machine (NVIDIA GeForce RTX 3060) which proved to be consistent and precise enough to see impact of the changes in the code. Evaluation was then performed by in-script implemented function with launching with a flag `./cuda --validate`.

Evaluation and profiling was done for 3 and 4 hits as only those were observed to be present. 5 hits were fitted as 4.

Base Data

As code was examined at before any changes done by me.

Description	Percentage of Total Time	Execution Time (μs)
`kernelBLFit<(int)4>` - BLF4	17.7%	94.9
`kernelBLFit<(int)3>` - BLF3	9.8%	76.7

Table 1: Base performance data for PixelTriplets Broken Line algorithm.

With Cooperative Groups - For Loops Only

After parallelization of `forloops` with cooperative group as described above. For matmul EIGEN was used and the multiplication was performed either by all threads in the group or only the first one.

As can be seen in table 2 there was improvement observed for hits of 4 but performance severely worsened for hits of 3. This is because for both, division to thread groups of 4 was used and in the case of 3 hits, one thread does no work throughout the whole fitting process.

Description	Percentage of Total Time	Execution Time (μs)
`kernelBLFit<(int)4>` - BLF4	12.2%	71.4
`kernelBLFit<(int)3>` - BLF3	22.5%	211.6

Table 2: Performance with cooperative groups for parallelized for-loops.

With Cooperative Groups - Including Matrix Multiplication

Own written parallel matmul routine was introduced. It was mainly dealing with coordinate system transformation A = J*A*J^T. As this involves effectively two matmul operations, two different approaches were taken. First multiplying them in two steps, meaning double nested for loops for i and j indexes of resulting matrix or one step triple nested for loop, with additional k index.

First showed to be faster and therefore only this is included in table. Improvement was mainly observed for 3 hits.

Description	Percentage of Total Time	Execution Time (μs)
2-step function (two simple for-loops)		
`kernelBLFit<(int)4>` - BLF4	12.1%	71.3
`kernelBLFit<(int)3>` - BLF3	21.7%	196.4

Table 3: Performance with cooperative groups including matrix multiplication.

With Cooperative Groups - Shared inputs and Matrix Multiplication

Effect of residing inputs in shared memory was also examined. As proposed by my supervisor, but asked to be confirmed, no positive effect was observed as from some point in NVIDIA chip development L2 cache and shared memory are hosted by the same hardware. Therefore increasing amount of shared memory needed lessens the amount available to be used as L2 cache. Therefore using excessive amount of shared memory is not recommended.

Description	Percentage of Total Time	Execution Time (μs)
Global memory for inputs		
`kernelBLFit<(int)4>` - BLF4	12.1%	71.3
`kernelBLFit<(int)3>` - BLF3	21.7%	196.4
Shared memory for inputs		
`kernelBLFit<(int)4>` - BLF4	12.1%	71.5
`kernelBLFit<(int)3>` - BLF3	21.6%	198.1

Table 4: Performance with cooperative groups including shared inputs.

With Cooperative Groups - cuBLASDx for matrix multiplication

Usage was nearly identical as mentioned in first section, although now $32/SizeOfGroup$ matrices multiplications were performed by the whole block instead of 32. Therefore less fits were stalled. However, this approach was performing the worse out of all matmul approaches.

Description	Percentage of Total Time	Execution Time (μs)
`kernelBLFit<(int)4>` - BLF4	13.4%	84.3
`kernelBLFit<(int)3>` - BLF3	25.2%	249.6

Table 5: Performance with and without cooperative groups.

SoA Layout Comparison

Standard SoA stands for $StructureofArrays$, which was used for all the other parts (array of individual matrixes). Although in CMSSW improved SoA layout is implemented and is proclaimed to maximize efficiency. For that reason same approach was examined here. Improvement is inherited from anticipated memory access to first element of any array within the structure first, followed by second element etc.. To gain leverage access pattern needs to be similar for all individual threads. Improved SoA layout is graphically depicted in fig. 1.

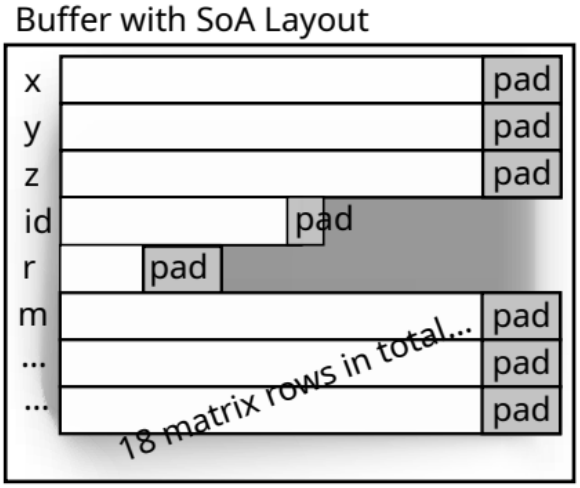

Figure 1: Improved SoA Layout in CMSSW

Description	Percentage of Total Time	Execution Time (μs)
Normal SoA		
`kernelBLFit<(int)4>` - BLF4	12.1%	55.5
`kernelBLFit<(int)3>` - BLF3	21.4%	221.8
Improved SoA		
`kernelBLFit<(int)4>` - BLF4	12.2%	61.6
`kernelBLFit<(int)3>` - BLF3	21.4%	222.3

Table 6: Performance comparison between normal SoA and improved SoA layout.

Operations - First Thread vs All Threads

Some of the operations were possible to be performed by either all or only the first thread e.g. writing results in global memory, operations on variables which could not be parallized and etc. In theory, only first thread approach should be faster as no memory collisions of threads reading/writing to same address should occure. Additional overhead created by if statement testing for the *thread id* within the groups was shown to counter act the effect, so no improvement was observed.

Description	Performance Difference
Operations done by first thread	No difference
Operations done by all threads	No difference

Table 7: Performance comparison between operations done by first thread vs all threads.

Connecting Two Kernels (FastFit and BLFit)

As launching of the kernel is creating an overhead. When division of algorithm in two kernels - FastFit and BLFit was observed, connecting them into one was natural way to improve the performance. This was proved to have a positive impact.

Description	Execution Time (sum of medians) (μs)
Normal	
`kernelBLFit<(int)4>` - BLF4	94.9 + 9.1 = 104.0
`kernelBLFit<(int)3>` - BLF3	76.7+8.6 = 85.3
Connected	
`kernelBLFit<(int)4>` - BLF4	94.3
`kernelBLFit<(int)3>` - BLF3	82.3

Table 8: Performance comparison between normal and connected kernels.

Combination of Kernels (Size 3 Normal, Size 4 Cooperative Groups with Matrix Multiplication)

From previously presented is clear that for 3 hits one not working thread in a group creates a huge disadvantage for cooperative groups. Therefore combination of one thread per fit for 3 hits and cooperative groups for 4 hits was examined. This was observed to improve significantly speed of fit on my machine.

Description	Percentage of Total Time	Execution Time (μs)
`kernelBLFit<(int)4>` - BLF4	13.9%	69.8
`kernelBLFit<(int)3>` - BLF3	8.9%	77.7

Table 9: Performance with a combination of normal and cooperative group kernels.

2.2 Performance examination - Online machine

Below is a summary of the throughput measurements performed on the ‘gpu-c2a02-39-03‘ machine, comparing the original implementation with the new one using Cooperative Groups. Boundary conditions for testing were- running on 8 threads, warmup of 10 000 events, 300 000 evaluated and this all repeated twenty times for approaches which showed to be faster locally (connecting kernels and combination of cooperative groups for 4 hits and one thread per fit for 3 hits).

After, average without a first measurement (apparent outlier) and std. dev. were calculated. Surprisingly no improvement was observed on online machine.

Configuration	Mean Throughput (events/sec)	Standard Deviation
Original	1080.9	7.2
Combination of Kernels	1050.5	9.1
Single Kernel	1077.8	14.1

Table 10: Throughput performance for different kernel configurations.

Application to CMSSW - FNNLS

Introduction

Use of cooperative groups in the Fast Non-Negative Least Squares (FNNLS) algorithm in the CMSSW ECAL reconstruction code was observed as part of an effort to explore its performance characteristics on GPUs. Unlike the PixelTracker standalone code, the FNNLS problem in ECAL reconstruction presents different challenges, particularly in memory management and parallelization with high divergence of threads due to a different sizes of for loops originating from the algorithms nature.

Memory Management

In contrast to the PixelTracker's fixed problem size, the FNNLS algorithm deals with for loops whose sizes are not fixed, varying dynamically during the execution. The typical matrix size observed was 10×10, but the algorithm iterates over actively and passively constrained variables, which change throughout the minimization process. Matrix size multiplied by the number of different variables needed throughout the algorithm necessitated the use of dynamic shared memory, as static shared memory was insufficient to accommodate the problem's requirements.

Parallelization and Cooperative Groups

Unlike PixelTracker, where cooperative groups could be leveraged for parallel execution, the FNNLS algorithm did not benefit significantly from such an approach. The inherent serial nature of the algorithm's key operations, such as Cholesky decomposition, its updates, and forward substitution, limited the effectiveness of parallelization. Although certain sums could theoretically be parallelized using atomic operations (e.g., `atomicAdd`), this approach essentially serializes the process due to the atomic nature of the operation. As a result, threads often remained idle, leading to inefficient utilization of GPU resources.

AtomicMax for Floating Point

An interesting problem encountered during this implementation was the lack of a native `atomicMax` function for floating-point numbers. While NVIDIA provides an approach using `atomicCAS` (Compare And Swap), a more efficient method was identified. This method involved representing the floating-point number as an integer, resulting in a performance improvement by approximately 100x.

`atomicCAS` approach:

```
__device__ static float atomicMaxFloat(float* address, float val)
{
    int* address_as_i = (int*) address;
    int old = *address_as_i, assumed;
    do {
        assumed = old;
        old = ::atomicCAS(address_as_i, assumed,
            __float_as_int(::fmaxf(val, __int_as_float(assumed))));
    } while (assumed != old);
    return __int_as_float(old);
}
```

`atomicMax` approach:

```
__device__ __forceinline__ float atomicMaxFloat (float * addr, float value)
        {
            float old;
            old = (value >= 0) ? __int_as_float(atomicMax((int *)addr,
                                    __float_as_int(value))) :
                  __uint_as_float(atomicMin((unsigned int *)addr,
                  __float_as_uint(value)));

            return old;
        }
```

As not only a value of the maximum (in this case error of the minimization process) but also index of corresponding element needed to be stored, two approaches emerged. First was using long integer like type consisting of float as first 4 bytes (representing the max value) and integer as the last 4 bytes (representing the index). Second approach was to perform the `atomicMax` two times as it returns the old value of the shared variable of max. After comparison of value of the thread and returned value is performed. If same index is written. Synchronization and reading the value is not possible as some threads may have already be gone out of the for loop. Both solutions shows promise and will be further investigated in the context of the Alpaka version of the same algorithm, which uses `atomicCAS`.

Performance Analysis

The overall performance of the FNNLS implementation in the ECAL reconstruction was worse compared to original implementation. Profiling revealed that the GPU kernel was approximately five times slower. Additionally, the overall process saw a 5% slowdown. These results highlight the challenges in optimizing such algorithms for GPU execution, particularly when dealing with inherently serial tasks.

Performance was evaluated with profiling by `NSIGHT COMPUTE` and timing of the whole process was done simply by Linux `time`.

Metric	Implementation	Percentage (%)	Average (ns)	Median (ns)
Kernel Minimize	Original	33.2%	278,510.3	261,358.5
	Only 1st Thread	36.3%	345,724.8	317,615.0
	Cooperative Groups	71.2%	1,767,504.6	1,208,889.0

Table 11: Performance metrics comparison for the `kernel_minimize` function across different implementations.

Implementation	Real Time (s)	User Time (s)	Sys Time (s)
Original	21.085	39.312	13.087
Only 1st Thread	21.330	37.998	12.810
Cooperative Groups	22.719	30.055	10.046

Table 12: Overall execution time comparison across different implementations.

Conclusion

The implementation of cooperative groups in context of FNNLS in the CMSSW ECAL reconstruction code revealed several challenges. The dynamic nature of the problem size and the serial characteristics of key operations limited the effectiveness of cooperative groups and parallel execution. Despite these challenges, the exploration of efficient atomic operations for floating-point numbers provided valuable insights for future work.