ACCELERATOR BASED PROGRAMMING
UPPSALA UNIVERSITY
FALL 2023

ASSIGNMENT 2: PROGRAMMING IN CUDA

Due: September 22nd. Upload your code and a report on Studium.

1. Benchmark description. For this assignment, we work with matrix-vector and matrix-matrix multiplications. All experiments are done with single-precision numbers, i.e., the C++ type `float`. The CPU code for the matrix-vector product of an $M \times N$ matrix A (M rows, N columns) with a vector x of size N, producing a vector y of size $M$is

```
for (int row = 0; row < M; ++row)
    {
        y[row] = 0.f;
        for (int col = 0; col < N; ++col)
        y[row] += A[row + M * col] * x[col];
    }
```

As you can see, we assume the matrix A to be stored in column-major order (data in neighboring rows of the same column stored consecutively). Your goal is write an algorithm that implements this and some related operations in CUDA. Note that a reduction is happening `y[row] += A[row + M * col] * x[col];` and your insights form Exercise 1 may come in useful.

Secondly, we extend the setting to matrix-matrix multiplication. Instead of a single vector x, we now assume an $N \times K$ matrix `B`, storing the result in an $M \times K$ matrix `C`:

```
for (int i = 0; i < M; ++i)
    for (int k = 0; k < K; ++k)
        {
            C[i + M * k] = 0.f;
            for (int j = 0; j < N; ++j)
            C[i + M * k] += A[i + j * M] * B[j + k * N];
        }
```

To measure the performance, we use the following metrics:

- Matrix-vector product: Achieved memory throughput in GB/s. Count the memory transfer for all matrix entries and for both vectors.
- Matrix-matrix product: Achieved floating point performance in GFlop/s. Computed as $MNK2$, the number of operations in the three nested loops.

Tasks

1. Write a program that implements the matrix-vector product by parallelizing over the matrix rows with CUDA code, starting e.g. from the stream triad cuda.cu program from assignment 1. Make sure to think about the appropriate memory allocation and initialization. Then, provide the following results and discussions:
   (a) Measure the achieved performance in GB/s for M = N with number of rows between 100 and 10000. Report results of at least 30 samples spread over the interval of interest.
   (b) Explain the expected performance by the available parallelism in terms of size of the thread blocks and the blocks per grid. What are good values for these two quantities?
2. As a point of comparison, we also want to compute the matrix-vector product with cuBLAS, the BLAS implementation provided by NVIDIA. To study the use cuBLAS, consult the page https://docs.nvidia.com/cuda/cublas/index.html.

how should we set vector? use kernel function or initialize in host and memorycpy to device?

We need to make the following additions to the code:

```
// ... other include files ...
#include "cublas_v2.h"

// in benchmark function
{
    cublasHandle_t handle;
    cublasStatus_t stat = cublasCreate(&handle);
    if (stat != CUBLAS_STATUS_SUCCESS)
        {
            std::cout << "CUBLAS initialization failed\n";
            std::abort();
        }

    float alpha = 1.f;
    float beta = 0.;
    stat =cublasSgemv(handle, CUBLAS_OP_N, M, N, &alpha,
                            mat, M, x, 1, &beta, y, 1);
    if (stat != CUBLAS_STATUS_SUCCESS)
        {
            std::cout << "CUBLAS operation failed\n";
            std::abort();
        }
    // ... maybe do some measurements or repetitions
    cublasDestroy(handle);
}
```

especially good for small size matrix, for large size there are too much overhead

In this code, the factors α and β correspond to the operation $y = \alpha Ax + \beta y$ and we use the setting to simply compute $y = Ax$. The second argument indices whether A should be used in non transposed (`CUBLAS_OP_N`) or transposed (`CUBLAS_OP_T`) form, the latter corresponding to $A^T x$. The variable of type `cublasHandle_t` is used to initialize and control the use of the cuBLAS functions and is usually only needed once per run. To compile the code with cuBLAS, we need to make sure to also link against the cuBLAS library, which we do by the compile command

```
nvcc -lcublas matrix-vector.cu -o matrix-vector
```

(a) Measure the performance of the cuBLAS version with similar code as above for $N = M$ and compare. Discuss possible performance differences (you do not need to reach the same performance with your code).

(b) Run the code with fixed N = 10000 and M modified as above. Record the performance of your code and cuBLAS.

(c) Run both variants with fixed M = 16384 and N between 10 and 1000. Discuss the performance differences.

3. We now want to compute $A^T x$ for $M = N$ with CUDA. How would you proceed in terms of parallelization? Think about a suitable thread layout and possible race conditions. Use your insights from the lab on the sum reduction (exercise 1). Compare the performance of a simple implementation for $M = N = 5000$ between your version and cuBLAS. it should give worse performance, my transpose one is not correct

4. The final step is to compute the matrix-matrix product. Write a basic version (no need to tune for high performance) and measure the performance for $M = N = K$ between 100 and 5000, using at least 20 samples. Compare to the performance of the naive CPU code (see above), and to using cuBLAS with the function call (using N for all dimensions):

```
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N,&alpha, A, N, B, N, &beta, C, N);
```