

IMPROVING PERFORMANCE OF THE MATRIX-MULTIPLICATION ALGORITHM WITH CUDA

Enrico Martini
VR406823

Michele Boldo
VR406589

2019

Contents

1	Devices Description	3
1.1	CPU	3
1.2	GP-GPU	3
2	Gaussian matrix multiplication algorithm	3
3	Matrix multiplication kernels	4
3.1	Sequential Matrix Multiplication	4
3.2	Parallel matrix multiplication without shared memory	4
3.3	Parallel matrix multiplication with shared memory	5
3.4	Parallel matrix multiplication with streams	6
4	Performance Analisis	9
4.1	Host code	9
4.2	Parallel code without shared memory and tiles	10
4.3	Parallel code with tiles and shared memory	11
4.4	Parallel code with streams	12
5	Graphic Analysis	13

1 Devices Description

1.1 CPU

To get a true and reliable final comparison 2 different CPUs were used. The AMD A8 7600 is installed in a server-pc with 16 Gb of RAM. On the other hand, HMP Dual Denver is the CPU of Jetson TX2, an embedded development platform equipped with all the features for AI Computing, with 8 Gb of RAM. Here is some other details:

Name	Cores	Max frequency
AMD A8 7600	4	3.8 GHz
HMP Dual Denver + Quad ARM	6	2.5 GHz

1.2 GP-GPU

For running the parallel code, 4 different GPUs were used. Geforce GTX 780, Geforce GTX 1060 and Tesla K40 are installed in 3 different server-pc. Tegra TX2 is the GPU of Jetson TX2 and the memory is in common between CPU and GPU. Here is some other details:

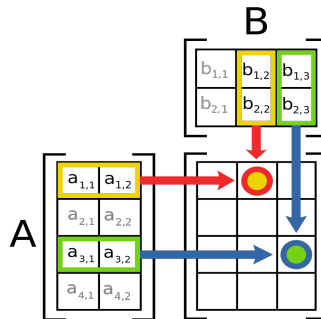
Comm. Name	Micro Arch.	CUDA cores	Memory	Frame Buffer	Compute Capability
GeForce GTX 780	Kepler	2304	3 Gb	900 MHz	3.5
GeForce GTX 1060	Pascal	1280	6 Gb	1708 MHz	6.1
Tesla K40	Kepler	2880	12 Gb	875 MHz	3.5
Tegra TX2	Pascal	256	8 Gb	1866 MHz	6.2

2 Gaussian matrix multiplication algorithm

Definition: let A an $n \times m$ matrix and B an $m \times p$ matrix, the matrix product $C = AB$ is defined to be the $n \times p$ matrix such that:

$$c_{ij} = \sum_{k=1}^m a_{ik} \cdot b_{kj}$$

Matrix multiplication shares some properties with usual multiplication. However, matrix multiplication is not defined if the number of columns of the first factor differs from the number of rows of the second factor and it is non-commutative.



3 Matrix multiplication kernels

List of functions of Gaussian matrix multiplication used for tests, written in C++ and CUDA.

3.1 Sequential Matrix Multiplication

```
void MatrixMulHost(double *A, double *B, double *C) {
    int c, d, k;
    for (c = 0; c < DIM; c++) {
        for (d = 0; d < DIM; d++) {
            int Pvalue = 0;
            for (k = 0; k < DIM; k++) {
                Pvalue += A[c * DIM + k] * B[k * DIM + d];
            }
            C[c * DIM + d] = Pvalue;
        }
    }
}
```

3.2 Parallel matrix multiplication without shared memory

```
--global--
void matrixMul(double *A, double *B, double *C, int size) {

    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    double Cvalue = 0.0;

    for (int k = 0; k < size; k++) {
        Cvalue += A[size * Row + k] * B[size * k + Row];
    }

    C[Row * size + Col] = Cvalue;
}

void addKernel(double *h_A, double *h_B, double *h_C, int size) {
    int size_tot = size * size * sizeof(double);
    double *d_A, *d_B, *d_C;

    cudaMalloc((void **)&d_A, size_tot);
    cudaMalloc((void **)&d_B, size_tot);
    cudaMalloc((void **)&d_C, size_tot);

    cudaMemcpy(d_A, h_A, size_tot, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size_tot, cudaMemcpyHostToDevice);

    dim3 block(16, 16, 1);
    dim3 grid((int)ceil((double)DIM / 16), (int)ceil((double)DIM / 16), 1);

    matrixMul << <grid, block >> > (d_A, d_B, d_C, size);

    cudaMemcpy(h_C, d_C, size_tot, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

3.3 Parallel matrix multiplication with shared memory

```
--global--
void MatrixMulKernelTiled(double *M, double *N, double *P, int size) {

    __shared__ double Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ double Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    int Pvalue = 0;

    for (int ph = 0; ph < (int)ceil(size / (double)TILE_WIDTH); ++ph) {
        if ((Row < size) && ((ph*TILE_WIDTH + tx) < size)) {
            Mds[ty][tx] = M[Row * size + ph * TILE_WIDTH + tx];
        }
        if (((ph * TILE_WIDTH + ty) < size) && (Col < size)) {
            Nds[ty][tx] = N[(ph*TILE_WIDTH + ty) * size + Col];
        }
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k) {
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();
    }

    if ((Row < size) && (Col < size)) {
        P[Row * size + Col] = Pvalue;
    }
}

void LaunchKernel(double *M, double *N, double *P, int size) {

    double *d_A, *d_B, *d_C;
    int spazio_tot = (size * size) * sizeof(double);
    cudaMalloc((void **)&d_A, spazio_tot);
    cudaMalloc((void **)&d_B, spazio_tot);
    cudaMalloc((void **)&d_C, spazio_tot);

    cudaMemcpy(d_A, M, spazio_tot, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, N, spazio_tot, cudaMemcpyHostToDevice);

    dim3 block(TILE_WIDTH, TILE_WIDTH, 1);
    dim3 grid(ceil((double)DIM / TILE_WIDTH), ceil((double)DIM /
        TILE_WIDTH), 1);

    MatrixMulKernelTiled << <grid, block >> > (d_A, d_B, d_C, size);

    cudaMemcpy(P, d_C, spazio_tot, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

3.4 Parallel matrix multiplication with streams

```
--global--
void MatMul(double* A, double* B, double* C, int ARows, int ACols, int BRows,
            int BCols, int CRows, int CCols) {

    double CValue = 0;
    int Row = blockIdx.y*TILE_DIM + threadIdx.y;
    int Col = blockIdx.x*TILE_DIM + threadIdx.x;

    __shared__ double As[TILE_DIM][TILE_DIM];
    __shared__ double Bs[TILE_DIM][TILE_DIM];

    for (int k = 0; k < (TILE_DIM + ACols - 1)/TILE_DIM; k++) {

        if (k*TILE_DIM + threadIdx.x < ACols && Row < ARows)
            As[threadIdx.y][threadIdx.x] = A[Row*ACols + k*
            TILE_DIM + threadIdx.x];
        else
            As[threadIdx.y][threadIdx.x] = 0.0;

        if (k*TILE_DIM + threadIdx.y < BRows && Col < BCols)
            Bs[threadIdx.y][threadIdx.x] = B[(k*TILE_DIM +
            threadIdx.y)*BCols + Col];
        else
            Bs[threadIdx.y][threadIdx.x] = 0.0;

        __syncthreads();

        for (int n = 0; n < TILE_DIM; ++n)
            CValue += As[threadIdx.y][n] * Bs[n][threadIdx.x];

        __syncthreads();
    }

    if (Row < CRows && Col < CCols)
        C[((blockIdx.y * blockDim.y + threadIdx.y)*CCols) +
        (blockIdx.x * blockDim.x) + threadIdx.x] = CValue;
}

void LaunchKernel(double *M, double *N, double *P, int size) {

    int n_stream = 4;
    cudaStream_t stream[n_stream];
    for(int i = 0; i < n_stream; i++)
        cudaStreamCreate(&stream[i]);

    double *N_3 = (double *)malloc(DIM * DIM/2 * (sizeof(double)));
    double *N_4 = (double *)malloc(DIM * DIM/2 * (sizeof(double)));

    for(int i = 0; i < DIM; i++){
        for(int j = 0; j < DIM; j++){
            if(j < DIM/2)
                N_3[i*DIM/2 + j]=N[i*DIM+j];
            else
                N_4[i*DIM/2 + (j-DIM/2)] = N[i*DIM + j];
        }
    }

    size_t slice = DIM * (DIM/2);
    double *d_M0, *d_N0, *d_P0;
    double *d_M1, *d_N1, *d_P1;
    double *d_M2, *d_N2, *d_P2;
    double *d_M3, *d_N3, *d_P3;
```

```

cudaMalloc((void **)&d_M0, DIM * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_N0, DIM * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_P0, (DIM/2) * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_M1, DIM * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_N1, DIM * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_P1, (DIM/2) * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_M2, DIM * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_N2, DIM * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_P2, (DIM/2) * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_M3, DIM * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_N3, DIM * (DIM/2) * sizeof(double));
cudaMalloc((void **)&d_P3, (DIM/2) * (DIM/2) * sizeof(double));

dim3 block(TILE_DIM, TILE_DIM, 1);
dim3 grid(ceil(((double)DIM / block.x), ceil(((double)DIM/2)/block.y)
, 1);

cudaMemcpyAsync(d_M0, M, slice * sizeof(double),
    cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(d_N0, N_3, slice * sizeof(double),
    cudaMemcpyHostToDevice, stream[0]);
double *ker0 = (double *)malloc(DIM/2 * DIM/2 * sizeof(double));

cudaMemcpyAsync(d_M1, M, slice * sizeof(double),
    cudaMemcpyHostToDevice, stream[1]);
cudaMemcpyAsync(d_N1, N_4, slice * sizeof(double),
    cudaMemcpyHostToDevice, stream[1]);
double *ker1 = (double *)malloc(DIM/2 * DIM/2 * sizeof(double));

cudaMemcpyAsync(d_M2, M + slice, slice * sizeof(double),
    cudaMemcpyHostToDevice, stream[2]);
cudaMemcpyAsync(d_N2, N_3, slice * sizeof(double),
    cudaMemcpyHostToDevice, stream[2]);
double *ker2 = (double *)malloc(DIM/2 * DIM/2 * sizeof(double));

cudaMemcpyAsync(d_M3, M + slice, slice * sizeof(double),
    cudaMemcpyHostToDevice, stream[3]);
cudaMemcpyAsync(d_N3, N_4, slice * sizeof(double),
    cudaMemcpyHostToDevice, stream[3]);
double *ker3 = (double *)malloc(DIM/2 * DIM/2 * sizeof(double));

MatMul<<<grid, block, block.x * block.y * sizeof(double), stream
[0]>>>(d_M0, d_N0, d_P0, DIM/2, DIM, DIM, DIM/2, DIM/2, DIM/2);
MatMul<<<grid, block, block.x * block.y * sizeof(double), stream
[1]>>>(d_M1, d_N1, d_P1, DIM/2, DIM, DIM, DIM/2, DIM/2, DIM/2);
MatMul<<<grid, block, block.x * block.y * sizeof(double), stream
[2]>>>(d_M2, d_N2, d_P2, DIM/2, DIM, DIM, DIM/2, DIM/2, DIM/2);
MatMul<<<grid, block, block.x * block.y * sizeof(double), stream
[3]>>>(d_M3, d_N3, d_P3, DIM/2, DIM, DIM, DIM/2, DIM/2, DIM/2);

cudaDeviceSynchronize();

cudaMemcpyAsync(ker0, d_P0, DIM/2 * DIM/2 * sizeof(double),
    cudaMemcpyDeviceToHost, stream[0]);
cudaMemcpyAsync(ker1, d_P1, DIM/2 * DIM/2 * sizeof(double),
    cudaMemcpyDeviceToHost, stream[1]);
cudaMemcpyAsync(ker2, d_P2, DIM/2 * DIM/2 * sizeof(double),
    cudaMemcpyDeviceToHost, stream[2]);
cudaMemcpyAsync(ker3, d_P3, DIM/2 * DIM/2 * sizeof(double),
    cudaMemcpyDeviceToHost, stream[3]);

for(int i = 0; i < DIM; i++){

```

```

        for(int j = 0; j<DIM ; j++){
            if(i < DIM/2 && j < DIM/2)
                P[i * DIM + j] = ker0[i * DIM/2 + j];
            else if(i < DIM/2 && j >= DIM/2)
                P[i * DIM + j] = ker1[i * DIM/2 + (j-DIM/2)];
            else if(i >= DIM/2 && j < DIM/2)
                P[i * DIM + j] = ker2[(i-DIM/2) * DIM/2 + j];
            else if(i >= DIM/2 && j >= DIM/2)
                P[i * DIM + j] = ker3[(i-DIM/2) * DIM/2 + (j-DIM/2)];
        }
    }

    cudaFree(d_M0);
    cudaFree(d_N0);
    cudaFree(d_P0);
    cudaFree(d_M1);
    cudaFree(d_N1);
    cudaFree(d_P1);
    cudaFree(d_M2);
    cudaFree(d_N2);
    cudaFree(d_P2);
    cudaFree(d_M3);
    cudaFree(d_N3);
    cudaFree(d_P3);
}

```


4 Performance Analysis

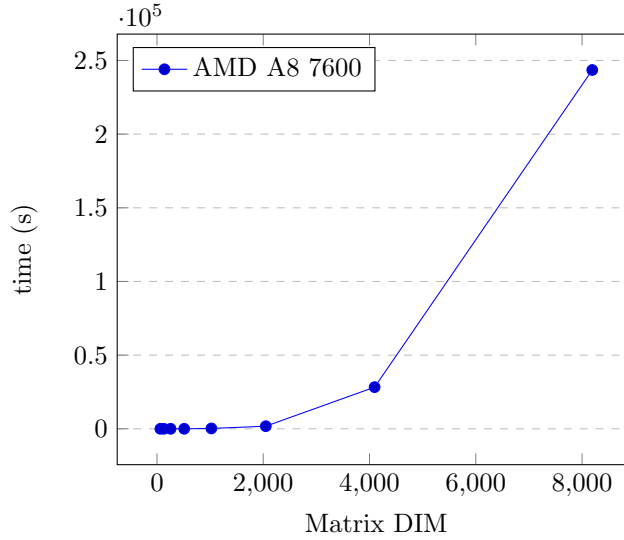
The performance test consists of a multiplication between square matrices using the canonical Gaussian algorithm. To have a uniformity of data, the matrices have been filled with standard values in each test. The values shown below are the average of three separate samples. All tests were performed in the same environment, with virtually no job active on the CPU/GPU. To improve the work load on devices and to check correctly the results, the data type of all the elements of the matrices are `double`.

4.1 Host code

Traditional sequential code on CPU.

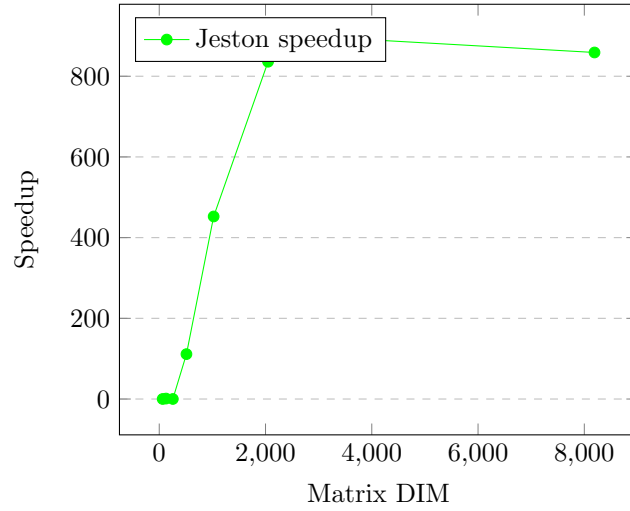
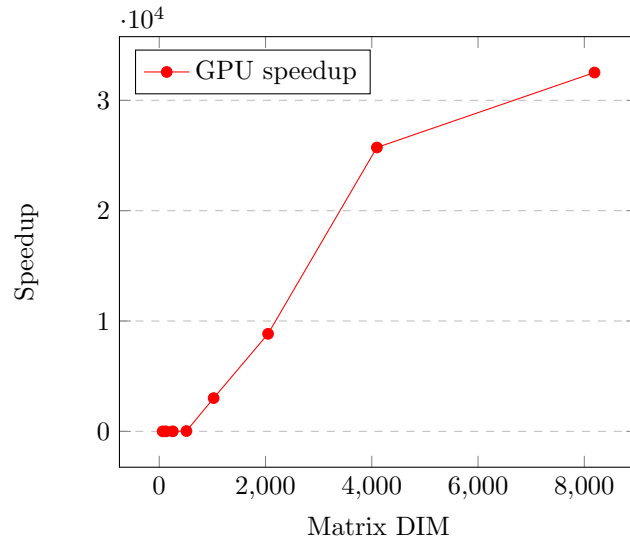
DIM	AMD A8 (s)	Jetson TX2 (s)
64	0.003	0.005
128	0.020	0.040
256	0.182	0.325
512	1.683	6.232
1024	239.950	77.364
2048	1758.773	877.728
4096	28265.020	7481.764
8192	243530.015	58063.970

Note: for 4k and 8k matrix in AMD A8 7600 the value is only a theoretical projection, due to the long time that those tasks could take.



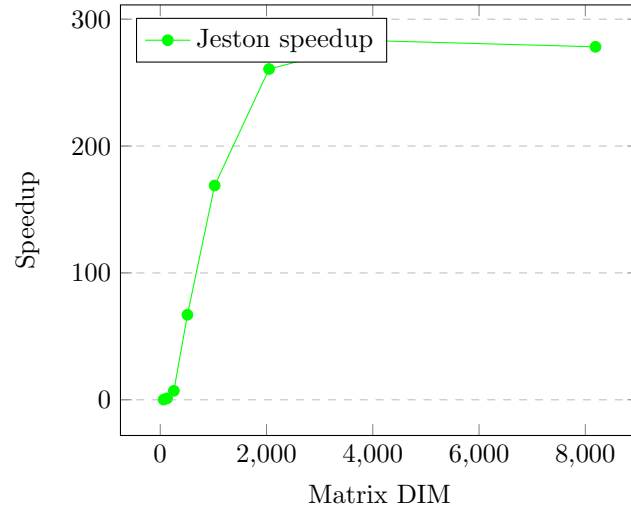
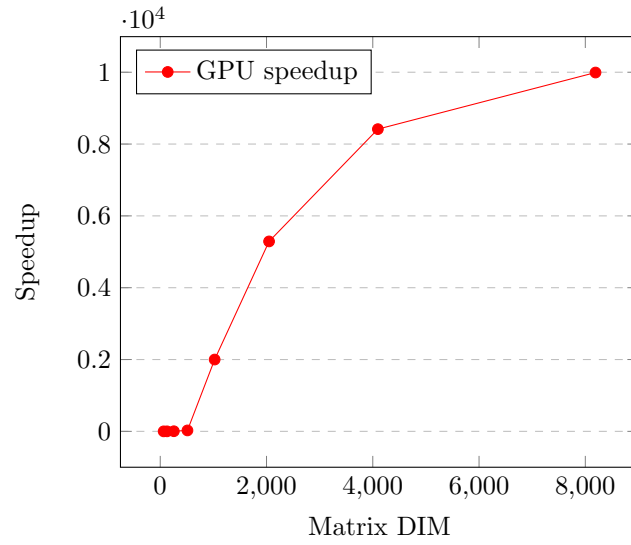
4.2 Parallel code without shared memory and tiles

DIM	GTX 780	GTX 1060	Jetson TX2	GPU Speedup	Jetson Speedup
64	0.383	0.080	0.038	0.04	0.13
128	0.344	0.060	0.035	0.34	1.14
256	0.374	0.060	0.040	3.06	0.13
512	0.360	0.063	0.056	26.71	111.28
1024	0.365	0.080	0.171	3011.92	452.42
2048	0.557	0.199	1.050	8836.58	835.93
4096	1.860	1.099	8.412	25726.66	889.42
8192	12.373	7.489	67.604	32519.81	858.869



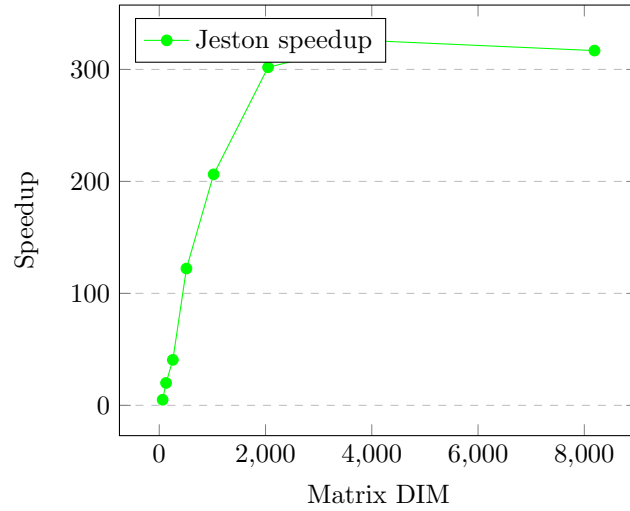
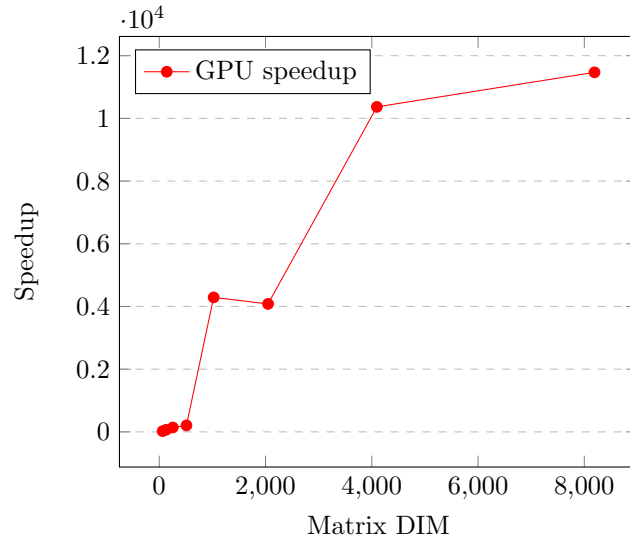
4.3 Parallel code with tiles and shared memory

DIM	GTX 780	GTX 1060	Jetson TX2	GPU Speedup	Jetson Speedup
64	0.324	0.087	0.036	0.03	0.14
128	0.355	0.067	0.038	0.30	1.14
256	0.333	0.054	0.046	3.38	7.06
512	0.355	0.064	0.093	26.15	67.01
1024	0.445	0.120	0.458	1999.58	168.91
2048	0.662	0.333	3.367	5289.01	260.69
4096	27.620	3.359	26.442	8415.55	282.95
8192	18.990	24.373	208.703	9991.66	278.21



4.4 Parallel code with streams

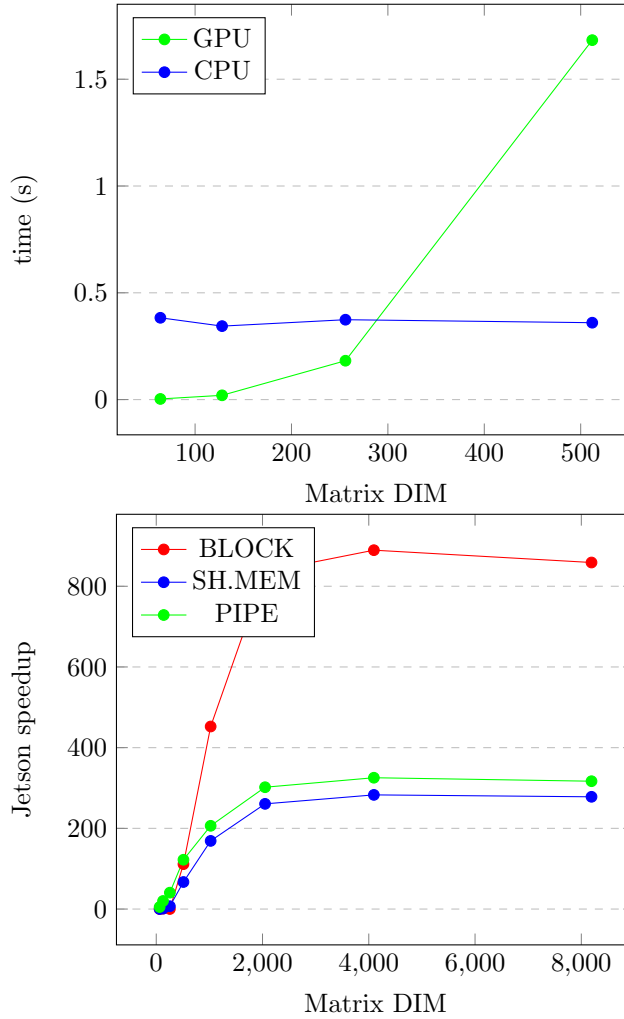
DIM	GTX 780	GTX 1060	Jetson TX2	GPU Speedup	Jetson Speedup
64	0.001	0.001	0.001	23.44	5.00
128	0.001	0.001	0.002	64.83	20.00
256	0.002	0.001	0.008	140.42	40.63
512	0.009	0.008	0.051	206.06	122.19
1024	0.049	0.056	0.375	4289.92	206.31
2048	0.294	0.431	2.907	4081.96	301.94
4096	1.943	2.726	22.989	10367.18	325.45
8192	14.687	21.233	183.258	11469.38	316.84



5 Graphic Analysis

In the first plot you can see that in multiplication between small matrices sequential code is more efficient than parallel code. The reason for this is the allocation and copy of data from host memory (RAM) to device memory (VRAM). This is a very expensive operation and makes worse performance on GPU. On the other hand, if the size of the matrices exceeds 512×512 , it is better to use parallel code because the time required to compute algebraic operations is significantly longer than the time of allocation and copy of the data.

Using a more advanced parallel code, which takes advantage of the use of shared memory and overlapping workloads, performance can be increased as much as possible, exploiting the full potential of the GP-GPU.



For the Jetson TX2, results are very different from the previous ones in some cases. As before, performance has improved greatly with the use of the GPU over sequential code. But it can be noticed that the most performing code is the code without tile and overlapping streams, in fact the architecture of this embedded board requires that CPU and GPU share the DRAM memory, so advanced parallel programming methods instead of speeding up the computation they slow it down.