MATRIX MULTIPLICATION SPEEDUP WITH CUDA (DOUBLE PRECISION)

Enrico Martini VR406823, Michele Boldo VR406589

SEQUENTIAL CODE

Device:

- AMD-A8 7600 (single core)

Code:

```
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;
        }
    }
}
```

Results (s):

MATRIX	TEST 1	TEST 2	TEST 3	MEAN
64	0.004	0.003	0.002	0.003
128	0.022	0.020	0.020	0.020
256	0.182	0.181	0.184	0.182
512	1.702	1.652	1.693	1.683
1024	238.646	240.346	240.857	239.950
2048	1756.849	1764.274	1755.197	1758.773
4096	28265.020	---	---	28265.020
8192	243530.020	---	---	243530.020

Note: for 4k and 8k matrix, the value is only a theorical proiection, due to the long time that those tasks could take.

PARALLEL CODE WITHOUT TILES

Device:

- NVIDIA GeForce GTX 780
- NVIDIA GeForce GTX 1060

Code:

```
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);
}
```

Results (s):

MATRIX	TEST 1	TEST 2	TEST 3	MEAN	SPEEDUP	DEVICE
64	0.396	0.378	0.376	0.383	0.01	GeForce 780
128	0.394	0.319	0.319	0.344	0.06	GeForce 780
256	0.357	0.451	0.313	0.374	0.49	GeForce 780
512	0.405	0.364	0.311	0.360	4.67	GeForce 780
1024	0.377	0.342	0.376	0.365	657.55	GeForce 780
2048	0.589	0.538	0.544	0.557	3157.97	GeForce 780
4096	1.860	1.860	1.860	1.860	15196.25	GeForce 780
8192	12.260	12.680	12.180	12.373	19681.84	GeForce 780

MATRIX	TEST 1	TEST 2	TEST 3	MEAN	SPEEDUP	DEVICE
64	0.131	0.055	0.053	0.080	0.04	GeForce 1060
128	0.074	0.053	0.053	0.060	0.34	GeForce 1060
256	0.072	0.054	0.053	0.060	3.06	GeForce 1060
512	0.075	0.057	0.057	0.063	26.71	GeForce 1060
1024	0.088	0.075	0.076	0.080	3011.92	GeForce 1060
2048	0.199	0.199	0.199	0.199	8836.58	GeForce 1060
4096	1.140	1.081	1.075	1.099	25726.66	GeForce 1060
8192	7.597	7.441	7.428	7.489	32519.81	GeForce 1060

PARALLEL CODE WITH TILES

Device:

- NVIDIA GeForce GTX 780
- NVIDIA GeForce GTX 1060

Code:

```
__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);

}
```

Results (s):

MATRIX	TEST 1	TEST 2	TEST 3	MEAN	SPEEDUP	DEVICE
64	0.322	0.334	0.317	0.324	0.01	GeForce 780
128	0.386	0.347	0.331	0.355	0.06	GeForce 780
256	0.319	0.343	0.337	0.333	0.55	GeForce 780
512	0.368	0.368	0.330	0.355	4.74	GeForce 780
1024	0.409	0.487	0.439	0.445	539.15	GeForce 780
2048	0.658	0.664	0.664	0.662	2657.38	GeForce 780
4096	27.270	28.390	27.200	27.620	1023.35	GeForce 780
8192	18.870	18.950	18.990	18.937	12860.24	GeForce 780

MATRIX	TEST 1	TEST 2	TEST 3	MEAN	SPEEDUP	DEVICE
64	0.127	0.079	0.056	0.087	0.03	GeForce 1060
128	0.068	0.077	0.056	0.067	0.30	GeForce 1060
256	0.054	0.054	0.054	0.054	3.38	GeForce 1060
512	0.063	0.067	0.063	0.064	26.15	GeForce 1060
1024	0.119	0.131	0.110	0.120	1999.58	GeForce 1060
2048	0.475	0.048	0.475	0.333	5289.01	GeForce 1060
4096	3.362	3.356	3.358	3.359	8415.55	GeForce 1060
8192	24.355	24.338	24.427	24.373	9991.66	GeForce 1060

PARALLEL CODE WITH STREAMS (v1)

Device:

- NVIDIA GeForce GTX 780
- NVIDIA GeForce GTX 1060

Code:

```
__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) {

    // Streams
    int n_stream = 4;
    cudaStream_t stream[n_stream];

    for(int i = 0; i < n_stream; i++){
        cudaStreamCreate(&stream[i]);
    }
```

```
    // Creation of 2 vertical sub-matrix
    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];
        }
    }

    // Vector allocation
    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);

    // Kernel Launch

    std::chrono::steady_clock::time_point start = std::chrono::steady_clock::now();

    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]);

    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    double tempo = std::chrono::duration_cast<std::chrono::duration<double> >(end - start).count();
    printf("%lf\n", tempo);

    // Creation of the final result matrix
    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);

}
```

Results (s):

MATRIX	TEST 1	TEST 2	TEST 3	MEAN	SPEEDUP	DEVICE
64	0.000	0.000	0.000	0.000	7.81	GeForce 780
128	0.001	0.001	0.001	0.001	31.34	GeForce 780
256	0.002	0.002	0.002	0.002	94.08	GeForce 780
512	0.009	0.009	0.009	0.009	193.76	GeForce 780
1024	0.049	0.049	0.049	0.049	4884.67	GeForce 780
2048	0.297	0.294	0.291	0.294	5980.19	GeForce 780
4096	1.950	1.950	1.930	1.943	14544.61	GeForce 780
8192	14.680	14.690	14.690	14.687	16581.71	GeForce 780

MATRIX	TEST 1	TEST 2	TEST 3	MEAN	SPEEDUP	DEVICE
64	0.000	0.000	0.000	0.000	23.44	GeForce 1060
128	0.000	0.000	0.000	0.000	64.83	GeForce 1060
256	0.001	0.001	0.001	0.001	140.42	GeForce 1060
512	0.008	0.008	0.008	0.008	206.06	GeForce 1060
1024	0.058	0.056	0.054	0.056	4289.92	GeForce 1060
2048	0.408	0.353	0.531	0.431	4081.96	GeForce 1060
4096	2.765	2.703	2.711	2.726	10367.18	GeForce 1060
8192	21.268	21.248	21.183	21.233	11469.38	GeForce 1060

PARALLEL CODE WITH STREAMS (v2)

Device:

- NVIDIA GeForce GTX 780
- NVIDIA GeForce GTX 1060

Code:

```
__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) {
    //Creazione Streams
    int n_stream = 4;
    cudaStream_t stream[n_stream];

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

    // Divisione della matrice in 2 sottomatrici
    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];
        }
    }

    //Allocazione dei segmenti di memoria
    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));


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

    // Esecuzione del kernel

    std::chrono::steady_clock::time_point start = std::chrono::steady_clock::now();

    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));
    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);

    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));
    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);


    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));
    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);

    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[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]);

    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    double tempo = std::chrono::duration_cast<std::chrono::duration<double> >(end - start).count();
    printf("%lf\n", tempo);

    // Copio le sottomatrici nella matrixce finale
    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);
}
```

Results (s):

MATRIX	TEST 1	TEST 2	TEST 3	MEAN	SPEEDUP	DEVICE
64	0.000	0.000	0.000	0.000	7.71	0.000
128	0.001	0.002	0.002	0.001	15.50	0.001
256	0.002	0.002	0.002	0.002	109.52	0.002
512	0.007	0.007	0.007	0.007	240.25	0.007
1024	0.042	0.041	0.042	0.042	5770.19	0.042
2048	0.264	0.259	0.266	0.263	6683.55	0.264
4096	1.850	1861.000	1.860	621.570	45.47	1.850
8192	14.260	14.250	14.260	14.257	17081.83	14.260
MATRIX	*TEST 1*	*TEST 2*	*TEST 3*	*MEAN*	*SPEEDUP*	*DEVICE*
64	*0.000*	*0.000*	*0.000*	*0.000*	*22.17*	*GeForce 1060*
128	*0.000*	*0.000*	*0.000*	*0.000*	*77.98*	*GeForce 1060*
256	*0.002*	*0.001*	*0.001*	*0.001*	*143.40*	*GeForce 1060*
512	*0.008*	*0.008*	*0.008*	*0.008*	*219.66*	*GeForce 1060*
1024	*0.051*	*0.056*	*0.536*	*0.214*	*1119.91*	*GeForce 1060*
2048	*0.340*	*0.340*	*0.340*	*0.340*	*5174.71*	*GeForce 1060*
4096	*2.654*	*2.662*	*2.664*	*2.660*	*10625.76*	*GeForce 1060*
8192	*21.141*	*21.110*	*21.113*	*21.121*	*11529.96*	*GeForce 1060*

<u>Time differeces between streams code v1 and streams code v2 on Geforce GTX 1060 (s):</u>

MATRIX	64	128	256	512	1024	2048	4096	8192	16384
V1	0.000126	0.000314	0.001299	0.008166	0.055933	0.430865	2.726393	21.23306	84.12907
V2	0.000135	0.000261	0.001272	0.00766	0.29331	0.339878	2.660047	21.12149	42.07339
SPEEDUP	0.945813	1.202806	1.021232	1.066014	0.190697	1.267703	1.024942	1.005282	1.999579