LP1 Assignment HPC H2

Parallel Computing using CUDA

Date - 31th August, 2020.

Assignment Number - HPC H2

Title

Parallel Computing using CUDA

Problem Definition

Vector and Matrix Operations-
Design parallel algorithm to

1. Add two large vectors
2. Multiply Vector and Matrix
3. Multiply two N × N arrays using n2 processors

Learning Objectives

- Learn parallel decomposition of problems.
- Learn parallel computing using CUDA

Learning Outcomes

I will be able to decompose problems into subproblems, to learn how to use GPUs, to learn to solve sub problems using threads on GPU cores.

Software Packages and Hardware Apparatus Used

- Operating System : 64-bit Ubuntu 18.04
- Browser : Google Chrome
- Programming Language : C++, Python 3
- Jupyter Notebook Environment : Google Colaboratory

Related Mathematics

Mathematical Model

Let S be the system set:

S = {s; e; X; Y; Fme;Ff;DD;NDD; Fc; Sc}

s=start state

e=end state

X=set of inputs
X = {X1,X2,X3,X4,X5,X6}
where X1, X2, X3 = Arrays
where X4, X5, X6 = Matrices

Y= Output Set
Y = {Y1,Y2,Y3} where
Y1 = X1 + X2
Y2 = X3 x X4
Y3 = X5 x X6

Fme is the set of main functions
Fme = {f0} where
F0 = output display function

Ff is the set of friend functions
Ff = {f1,f2,f3,f4,f5,f6,f7,f8} where

f1 = Kernel Function to add Arrays
f2 = Kernel Function to multiply array and matrix
f3 = Kernel Function to multiply 2 matrices

f4 = Host Function to initialize array/matrix
f5 = Host Function to display array/matrix (Overload insertion operator)
f6 = Host Function to add Arrays (Overload + operator)
f7 = Host Function to multiply array and matrix (Overload * operator)
f8 = Host Function to multiply 2 matrices (Overload * operator)

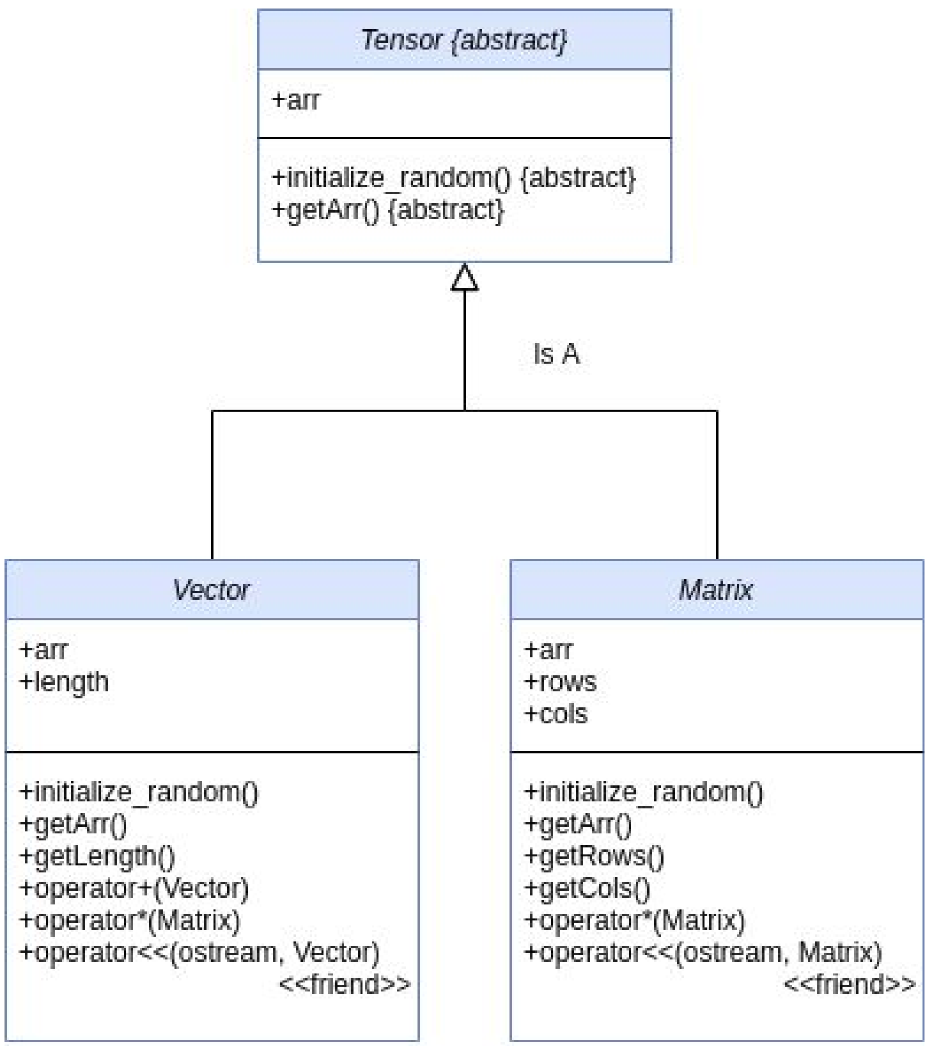
DD = Deterministic Data

Input Array X1,X2,X3
Input Matrices X4,X5,X6

NDD=Non-deterministic data
No non deterministic data

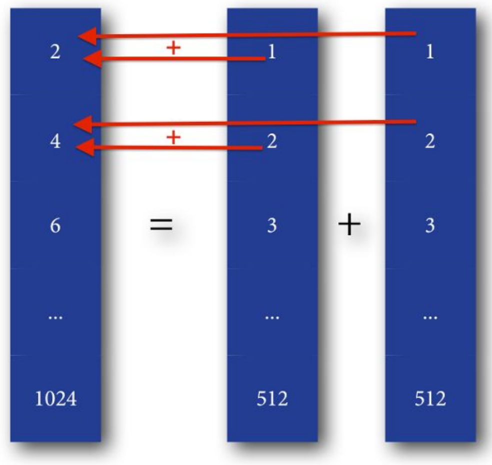
Fc =failure case:
No failure case identified for this application

Class Diagram

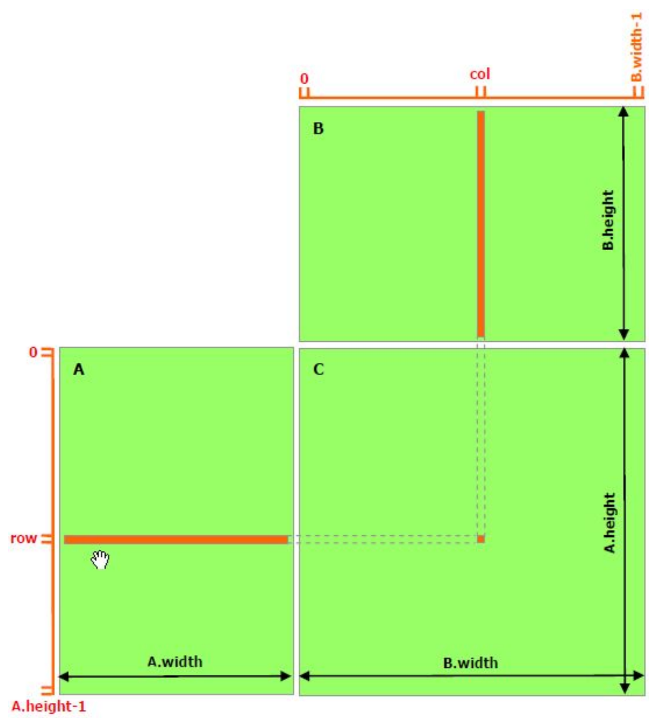


Diagrams

Array Addition



Matrix Multiplication



Vector Matrix Multiplication

$$\langle A|\alpha = \begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix} \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix}$$

$$= \begin{bmatrix} A_1\alpha_{11} + A_2\alpha_{21} + A_3\alpha_{31} & A_1\alpha_{12} + A_2\alpha_{22} + A_3\alpha_{32} & A_1\alpha_{13} + A_2\alpha_{23} + A_3\alpha_{33} \end{bmatrix}$$

Concepts related Theory

Dividing a computation into smaller computations and assigning them to different processors for parallel execution are the two key steps in the design of parallel algorithms.
The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel, is called decomposition.

Tasks are programmer-defined units of computation into which the main computation is subdivided by means of decomposition. Simultaneous execution of multiple tasks is the key to reducing the time required to solve the entire problem.

Tasks can be of arbitrary size, but once defined, they are regarded as indivisible units of computation. The tasks into which a problem is decomposed may not all be of the same size.
In addition of two vectors, we have to add ith element from first array with ith element of the second array to get ith element of the resultant array. We can allocate this each addition to a distinct thread. Same thing can be done for the product of two vectors.

There can be three cases for addition of two vectors using CUDA.
1. n blocks and one thread per block.
2. 1 block and n threads in that block.
3. m blocks and n threads per block.

In addition of two matrices, we have to add (i,j)th element from first matrix with (i,j)th element of the second matrix to get (i,j)th element of resultant matrix.. We can allocate this each addition to a distinct thread.

There can be two cases for addition of two matrices using CUDA.
1. Two dimensional blocks and one thread per block.
2. One block and two dimensional threads in that block.

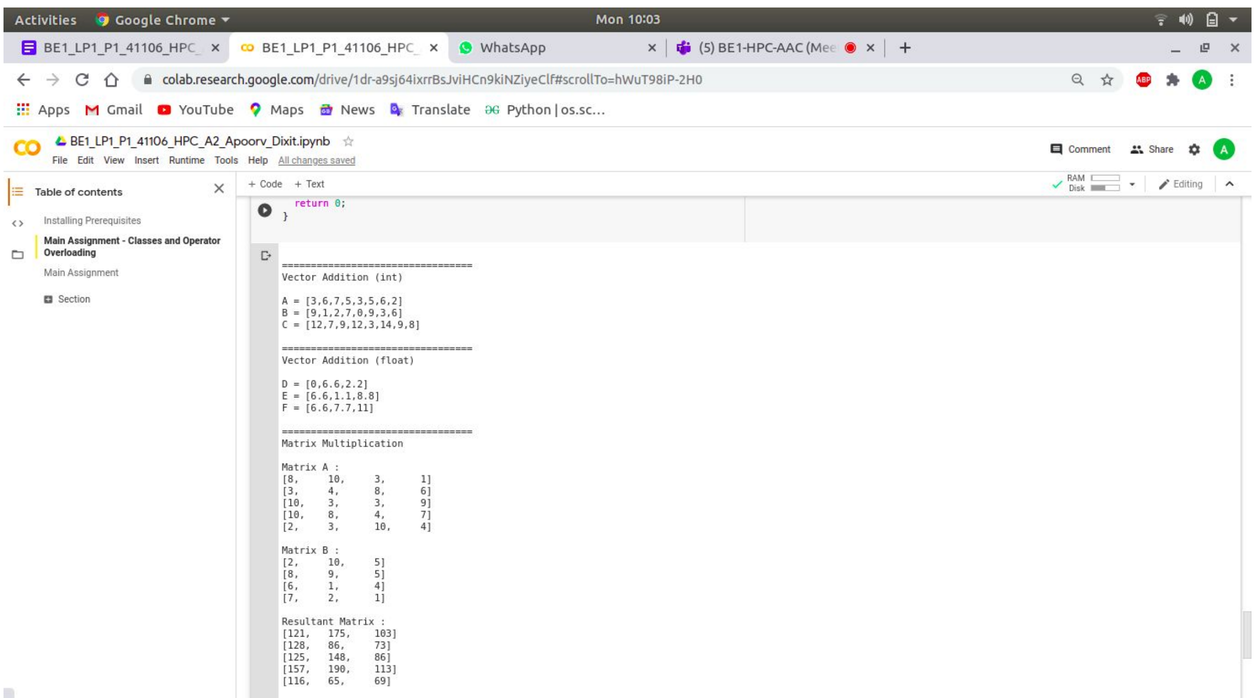
Same cases can be considered for multiplication of two matrices.

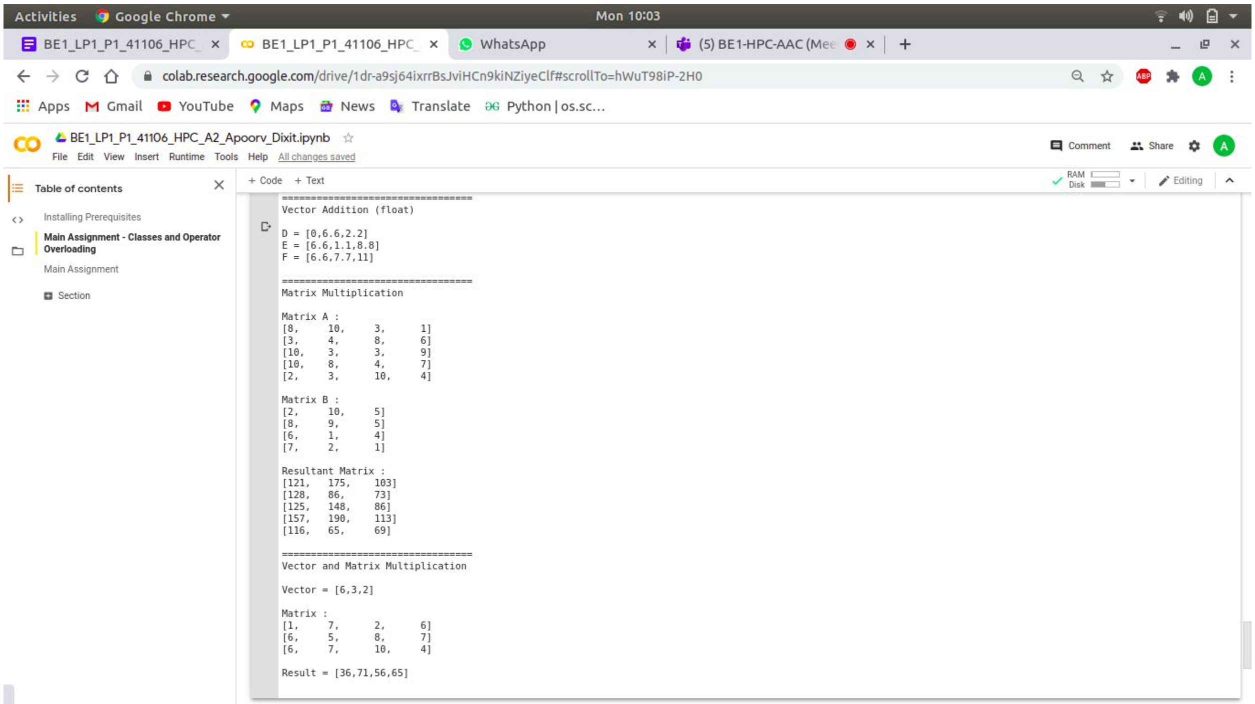
Google Colab Notebook Link

https://colab.research.google.com/drive/1dr-a9sj64ixrrBsJviHCn9kiNZiyeClf?usp=sharing

Output Screenshots

Get Array Addition, Product of Vector-Matrix and Matrices Product





Source Code

```
#include<iostream>
#include<cstdlib>
using namespace std;

//Kernel Functions
template <typename T>
__global__ void vectorAdd(T *a, T *b, T *result, int n) {
 int tid = blockIdx.x*blockDim.x + threadIdx.x;
 result[tid] = a[tid] + b[tid];
}

template <typename T>
__global__ void matrixMultiplication(T *a, T *b, T *c, int m, int n, int
k){
 int row = blockIdx.y*blockDim.y + threadIdx.y;
 int col = blockIdx.x*blockDim.x + threadIdx.x;
 T sum=0;
 if(col<k && row<m) {
```

```
   for(int j=0;j<n;j++){
     sum += a[row*n+j] * b[j*k+col];
   }
   c[k*row+col]=sum;
 }
}

template <typename T>
__global__ void matrixVector(T *vec, T *mat, T *result, int n, int m) {
 int tid = blockIdx.x*blockDim.x + threadIdx.x;
 T sum=0;
 for(int i=0; i<n; i++) {
   sum += vec[i]*mat[(i*m) + tid];
 }
 result[tid] = sum;
}

//Classes
template <class T>
class Tensor{
 public:
 virtual void initialize_random() const=0;
 virtual T* getArr() const=0;
};

template <class T>
class Matrix : public Tensor<T>{

 private:
 T* arr;
 int rows, cols;

 public:
 Matrix(int r, int c, bool init_rand=true){
   rows = r;
   cols = c;
   arr = new T[rows*cols];
```

```
  if(init_rand==true){
    initialize_random();
  }
}

void initialize_random() const {
  for(int i=0; i<rows; i++) {
    for(int j=0; j<cols; j++) {
      arr[i*cols + j] = rand()%10 + 1;
    }
  }
}

T* getArr() const {
  return arr;
}

int getRows() const {
  return rows;
}

int getCols() const{
  return cols;
}

Matrix operator* (const Matrix& arg){

  //Declaring Device Variables
  int *a_dev,*b_dev,*c_dev;
  int m=rows, n=cols, k=arg.cols;
  Matrix c(m,k,false);

  //Allocating Memory to Device Variables
  cudaMalloc(&a_dev, sizeof(T)*m*n);
  cudaMalloc(&b_dev, sizeof(T)*n*k);
  cudaMalloc(&c_dev, sizeof(T)*m*k);
```

```
    //Copying Mmmory from CPU to GPU (operands)
    cudaMemcpy(a_dev, arr, sizeof(T)*m*n, cudaMemcpyHostToDevice);
    cudaMemcpy(b_dev, arg.arr, sizeof(T)*n*k, cudaMemcpyHostToDevice);

    int dimb = (m>n)?m:n;
    dimb = (k>dimb)?k:dimb;
    dim3 dimGrid(1,1);
    dim3 dimBlock(dimb,dimb);
    matrixMultiplication<<<dimGrid, dimBlock>>>(a_dev,b_dev,c_dev, m, n,
k);

    cudaMemcpy(c.arr, c_dev, sizeof(T)*m*k, cudaMemcpyDeviceToHost);

    cudaFree(a_dev);
    cudaFree(b_dev);
    cudaFree(c_dev);

    return c;

  }

  //Overloading insertion operator to display Vector
  //Friend Function
  template <typename U>
  friend ostream& operator<<(ostream& os, const Matrix <U> &matrix);

};

template <typename T>
ostream& operator<<(ostream& os, const Matrix<T> &matrix){
  os<<endl;
  for(int i=0; i<matrix.rows; i++) {
    os<<"["<<matrix.arr[i*matrix.cols];
    for(int j=1; j<matrix.cols; j++){
      os<<",\t"<<matrix.arr[i*matrix.cols + j];
    }
    os<<"]"<<endl;
```

```
 }
 return os;
}


template<class T>
class Vector : public Tensor<T> {

 //Private Variables
 private:
 T *arr;
 int length;

 //Public Methods
 public:

 //Constructor with Default Parameter init_rand
 Vector(int n, bool init_rand=true){
   length = n;
   arr = (T*)malloc(n * sizeof(T));
   if(init_rand==true){
     initialize_random();
   }
 }

 //Initializes Elements to a random value
 void initialize_random() const{
   for(int i=0; i<length; i++) {
     arr[i] = rand()%10 * 1.1;
   }
 }

 //Returns number of array elements
 int getLength() const {
   return length;
 }
```

```
T* getArr() const {
  return arr;
}

//Overloading + operator for addition of two vectors
Vector operator+(const Vector& arg) {

  if(length!=arg.length){
    return Vector(0);
  }

  Vector result(length, false);

  //Device Variables
  T *this_dev, *arg_dev, *result_dev;

  int size = length * sizeof(T);

  //Allocating Memory to Device Variables
  cudaMalloc(&this_dev, size);
  cudaMalloc(&arg_dev, size);
  cudaMalloc(&result_dev, size);

  //Copying Variables from Host to Device
  cudaMemcpy(this_dev, arr, size, cudaMemcpyHostToDevice);
  cudaMemcpy(arg_dev, arg.arr, size, cudaMemcpyHostToDevice);

  //Kernel Launch
  vectorAdd<<<1,length>>>(this_dev, arg_dev, result_dev, length);

  //Copying Variables from Device to Host
  cudaMemcpy(result.arr, result_dev, size, cudaMemcpyDeviceToHost);

  //Freeing Device Memory
  cudaFree(this_dev);
  cudaFree(arg_dev);
  cudaFree(result_dev);
```

```
  return result;

}


//Overloading * operator for product of vector and matrix
Vector operator*(const Matrix<T> &arg) {

  int *a_dev, *b_dev, *c_dev;

  int n = length;
  int m = arg.getCols();

  Vector c(m,false);

  cudaMalloc(&a_dev, sizeof(int)*n);
  cudaMalloc(&b_dev, sizeof(int)*n*m);
  cudaMalloc(&c_dev, sizeof(int)*m);

  T* arg_arr = arg.getArr();
  cudaMemcpy(a_dev, arr, sizeof(int)*n, cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, arg_arr, sizeof(int)*n*m, cudaMemcpyHostToDevice);

  int dimt = (m>n)?m:n;

  matrixVector<<<1, dimt>>>(a_dev, b_dev, c_dev, n, m);

  cudaMemcpy(c.arr, c_dev, sizeof(int)*m, cudaMemcpyDeviceToHost);

  cudaFree(a_dev);
  cudaFree(b_dev);
  cudaFree(c_dev);

  return c;

}
```

```
 //Overloading insertion operator to display Vector
 //Friend Function
 template <typename U>
 friend ostream& operator<<(ostream& os, const Vector<U> &vec);

};

template <typename T>
ostream& operator<<(ostream& os, const Vector<T> &vec){
 if(vec.length==0){
   os<<"[ ]";
   return os;
 }
 os<<"["<<vec.arr[0];
 for(int i=1; i<vec.length; i++) {
   os<<","<<vec.arr[i];
 }
 os<<"]";
 return os;
}



int main(){
 cout<<"\n==================================\n";
 cout<<"Vector Addition (int)\n";

 //Vector of type int
 Vector<int> a(8), b(8), c=a+b;
 cout<<"\nA = "<<a;
 cout<<"\nB = "<<b;
 cout<<"\nC = "<<c;
 cout<<endl;
 cout<<"\n==================================\n";
 cout<<"Vector Addition (float)\n";
```

```
 //Vector of type float
 Vector<float> d(3), e(3), f=d+e;
 cout<<"\nD = "<<d;
 cout<<"\nE = "<<e;
 cout<<"\nF = "<<f;
 cout<<endl;
 cout<<"\n=================================\n";
 cout<<"Vector and Matrix Multiplication\n";
 Vector<int> v(3);
 Matrix<int> mat(3,4);
 Vector<int> res = v*mat;
 cout<<"\nVector = "<<v;
 cout<<"\n\nMatrix : "<<mat;
 cout<<"\nResult = "<<res;
 cout<<endl;
 cout<<"\n=================================\n";
 cout<<"Matrix Multiplication\n";

 int m=5, k=4, n=3;
 Matrix<int> m1(m,k),m2(k,n),m3=m1*m2;
 cout<<"\nMatrix A : "<<m1;
 cout<<"\nMatrix B : "<<m2;
 cout<<"\nResultant Matrix : "<<m3;

 return 0;
}
```

Strassen Algorithm - Matrix Multiplication (Modification)

Algorithm

Let A, B be two square matrices over a ring R. We want to calculate the matrix product C as

$$\mathbf{C} = \mathbf{A}\mathbf{B} \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

If the matrices A, B are not of type 2n × 2n we fill the missing rows and columns with zeros.

We partition A, B and C into equally sized block matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

with

$$\mathbf{A}_{i,j}, \mathbf{B}_{i,j}, \mathbf{C}_{i,j} \in R^{2^{n-1} \times 2^{n-1}}$$

The naive algorithm would be:

$$\mathbf{C}_{1,1} = \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1}$$

$$\mathbf{C}_{1,2} = \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2}$$

$$\mathbf{C}_{2,1} = \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1}$$

$$\mathbf{C}_{2,2} = \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}$$

With this construction we have not reduced the number of multiplications. We still need 8 multiplications to calculate the Ci,j matrices, the same number of multiplications we need when using standard matrix multiplication.

The Strassen algorithm defines instead new matrices:

$$\mathbf{M}_1 := (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2})$$

$$\mathbf{M}_2 := (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1}$$

$$\mathbf{M}_3 := \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2})$$

$$\mathbf{M}_4 := \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1})$$

$$\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2}$$

$$\mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$$

$$\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})$$

only using 7 multiplications (one for each Mk) instead of 8. We may now express the Ci,j in terms of Mk:

$$\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$

$$\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$$

$$\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$$

$$\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

We iterate this division process n times (recursively) until the submatrices degenerate into numbers (elements of the ring R). The resulting product will be padded with zeroes just like A and B, and should be stripped of the corresponding rows and columns.

Practical implementations of Strassen's algorithm switch to standard methods of matrix multiplication for small enough submatrices, for which those algorithms are more efficient. The particular crossover point for which Strassen's algorithm is more efficient depends on the specific implementation and hardware. Earlier authors had estimated that Strassen's algorithm is faster for matrices with widths from 32 to 128 for optimized implementations. However, it has been observed that this crossover point has been increasing in recent years, and a 2010 study found that even a single step of Strassen's algorithm is often not beneficial on current architectures, compared to a highly optimized traditional multiplication, until matrix sizes exceed 1000 or more, and even for matrix sizes of several thousand the benefit is typically marginal at best (around 10% or less). A more recent study (2016) observed benefits for matrices as small as 512 and a benefit around 20%.

Complexity

The standard matrix multiplication takes approximately $2N^3$ (where $N = 2^n$) arithmetic operations (additions and multiplications); the asymptotic complexity is $\Theta(N^3)$.

The number of additions and multiplications required in the Strassen algorithm can be calculated as follows: let $f(n)$ be the number of operations for a $2^n \times 2^n$ matrix. Then by recursive application of the Strassen algorithm, we see that $f(n) = 7f(n-1) + \ell 4^n$, for some constant ℓ that depends on the number of additions performed at each application of the algorithm. Hence $f(n) = (7 + o(1))^n$, i.e., the asymptotic complexity for multiplying matrices of size $N = 2^n$ using the Strassen algorithm is

$$O([7+o(1)]^n) = O(N^{\log_2 7+o(1)}) \approx O(N^{2.8074})$$

.

The reduction in the number of arithmetic operations however comes at the price of a somewhat reduced numerical stability and the algorithm also requires significantly more memory compared to the naive algorithm. Both initial matrices must have their dimensions expanded to the next power of 2, which results in storing up to four times as many elements, and the seven auxiliary matrices each contain a quarter of the elements in the expanded ones.

The "naive" way of doing the matrix multiplication would require 8 instead of 7 multiplications of sub-blocks. This would then give rise to the complexity one expects from the standard approach:

$$O(8^{log_2 n}) = O(N^{\log_2 8}) = O(N^3)$$

Conclusion

I have successfully divided problems into subproblems, learnt how to use GPUs and learnt how to solve sub problems using threads on GPU cores.