

Accelerate a simple C++ program with MPI and CUDA

This article will discuss parallel computing and how it can be applied using MPI and CUDA paradigms. We will use these paradigms to accelerate the multiplication of two large matrices in a simple C++ program.

Parallel computing refers to executing multiple tasks or processes simultaneously, dividing them among multiple computing resources such as processors or cores. This approach allows for the efficient utilization of resources and can significantly speed up the execution of certain types of problems.

Parallel computing is particularly well-suited for solving problems that can be divided into smaller, independent tasks that can be executed simultaneously. These problems often involve large datasets, complex calculations, or repetitive operations. By dividing the workload among multiple computing resources, parallel computing can significantly reduce the execution time.

Problems suitable for parallel computing include matrix multiplication, image and video processing, scientific simulations, data analysis, and machine learning algorithms. These problems typically involve extensive calculations that can be broken down into smaller tasks and executed in parallel.

Many models or paradigms offer different approaches to parallel computing, catering to various hardware architectures and programming requirements. Each has its strengths and is selected based on the specific needs of the application at hand. Here's a brief introduction to several paradigms commonly used in parallel computing:

1. **OpenMP:** OpenMP (Open Multi-Processing) is a popular shared-memory parallel programming model. It allows developers to parallelize their code by inserting directives that specify which parts of the code can be executed in parallel. OpenMP is known for its simplicity and ease of use, making it a widely adopted choice for parallel programming on multi-core processors.
2. **SIMD:** SIMD (Single Instruction, Multiple Data) is a parallel computing paradigm that focuses on executing the same operation on multiple data elements simultaneously. It is commonly used in vector processing architectures, where a single instruction operates on multiple data elements in a single clock cycle. SIMD provides significant performance improvements for tasks that involve data-level parallelisms, such as image processing and multimedia applications.
3. **MPI:** MPI (Message Passing Interface) is a standard communication protocol used for parallel computing across distributed memory systems. It enables multiple processes on different machines to exchange data and coordinate their execution. MPI is widely used

in scientific and high-performance computing applications, where data is distributed across multiple nodes and requires efficient communication and synchronization.

4. OpenCL: OpenCL (Open Computing Language) is an open standard for parallel programming across heterogeneous platforms, including CPUs, GPUs, and FPGAs. It allows developers to write code that can be executed on different hardware architectures, enabling high-performance computing across various devices. OpenCL provides a flexible and portable approach to parallel computing and is particularly useful for tasks that can benefit from the acceleration of specialized hardware.
5. CUDA: CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It allows developers to harness the power of NVIDIA GPUs for general-purpose computing. CUDA provides a programming interface and tools that enable developers to write code that can execute in parallel on GPUs, achieving significant performance improvements for various applications, including scientific simulations, data analytics, and deep learning.

Matrix multiplication

The [schoolbook algorithm](#), also known as the naive algorithm or the standard multiplication algorithm, is a method used to multiply matrices. To multiply two matrices, A and B, we iterate through each row of A and each column of B. For each element in the resulting matrix C, we multiply the corresponding row of A with the corresponding column of B, summing up the products to obtain the final value.

$$a_{2,1}b_{1,1} + a_{2,2}b_{2,1} + a_{2,3}b_{3,1}$$
$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} b_{1,1} \\ b_{2,1} \\ b_{3,1} \end{bmatrix} \begin{bmatrix} b_{1,2} & b_{1,3} \\ b_{2,2} & b_{2,3} \\ b_{3,2} & b_{3,3} \end{bmatrix} = \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \\ \square & \square & \square \end{bmatrix}$$

The schoolbook algorithm can be efficiently parallelized. Since each element of the resulting matrix C is calculated independently, it lends itself well to parallel execution. We can assign different subsets of rows or columns to different processors or threads, allowing them to work on their

respective parts simultaneously. Once all the calculations are complete, the partial results can be combined to obtain the final matrix C.

All the following source codes and the compiled binaries are stored in /pfss/toolkit/parallel-computing-cpp-example.

The single code way

```
#include <iostream>
#include <chrono>
#include <random>

using namespace std;

const int N = 1500; // matrix size

int main() {

    // allocate matrices dynamically
    float **A, **B, **C;
    A = new float*[N];
    B = new float*[N];
    C = new float*[N];
    for(int i = 0; i < N; i++) {
        A[i] = new float[N];
        B[i] = new float[N];
        C[i] = new float[N];
    }

    // create and initialize matrices
    default_random_engine generator;
    uniform_real_distribution<float> distribution(0.0,1.0);
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            A[i][j] = distribution(generator);
            B[i][j] = distribution(generator);
            C[i][j] = 0.0;
        }
    }
}
```

```

auto start = chrono::high_resolution_clock::now();

// matrix multiplication
for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
        for(int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
cout << "Time taken: " << duration.count() << " ms" << endl;

// free memory
for(int i = 0; i < N; i++) {
    delete[] A[i];
    delete[] B[i];
    delete[] C[i];
}
delete[] A;
delete[] B;
delete[] C;

return 0;
}

```

```

# load the gcc compiler from lmod
module load GCC/11.3.0

# compile the code
g++ mm-1core.cpp -o mm-1core

# run on a compute node with one CPU core
# the program will not run any faster even if we give it more cores
# because the program don't know how to utilize them
srun -p gpu -c1 ./mm-1core

```

The MPI way

```

#include <iostream>
#include <chrono>
#include <random>
#include <mpi.h>

using namespace std;

const int N = 1500; // matrix size

int main(int argc, char** argv) {
    auto start = chrono::high_resolution_clock::now();
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // create and initialize matrices
    float *A = new float[N*N];
    float *B = new float[N*N];
    float *C = new float[N*N];
    default_random_engine generator;
    uniform_real_distribution<float> distribution(0.0,1.0);
    for(int i = 0; i < N*N; i++) {
        A[i] = distribution(generator);
        B[i] = distribution(generator);
        C[i] = 0.0;
    }

    // distribute matrices A and B
    int chunk_size = N / size;
    float *local_C = new float[chunk_size*N];
    for(int i = 0; i < chunk_size*N; i++) {
        local_C[i] = 0.0;
    }

    MPI_Bcast(A, N*N, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(B, N*N, MPI_FLOAT, 0, MPI_COMM_WORLD);

    // matrix multiplication

```

```

const int i_offset = rank * chunk_size;

for(int i = 0; i < chunk_size; i++) {
    for(int j = 0; j < N; j++) {
        for(int k = 0; k < N; k++) {
            local_C[i*N+j] += A[(i + i_offset)*N+k] * B[(k + i_offset)*N+j];
        }
    }
}

// gather results from all processes
MPI_Gather(local_C, chunk_size*N, MPI_FLOAT, C,
           chunk_size*N, MPI_FLOAT, 0, MPI_COMM_WORLD);

// clean up
delete[] A;
delete[] B;
delete[] C;
delete[] local_C;

MPI_Finalize();

// print timing information
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
if(rank == 0) {
    cout << "Time taken: " << duration.count() << " ms" << endl;
}

return 0;
}

```

this time we need OpenMPI as well to compile our code

module load GCC/11.3.0 OpenMPI/4.1.4

compile with the mpic++ compiler

mpic++ mm-mpi.cpp -o mm-mpi

run the compiled program with 8 CPU cores

srun -n8 -N1 -p gpu ./mm-mpi

The CUDA way

```
#include <iostream>
#include <chrono>
#include <random>

using namespace std;

const int N = 1500; // matrix size
// divide the matrix into 94x94=8836 sub-matrix (blocks)
// each blocks has 16x16=256 threads taking care of each number in the result matrix
const int BLOCK_SIZE = 16;

// CUDA kernel for matrix multiplication
__global__ void matrixMul(float *A, float *B, float *C, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0;
    for(int k = 0; k < n; k++) {
        sum += A[row * n + k] * B[k * n + col];
    }
    C[row * n + col] = sum;
}

int main() {
    auto start = chrono::high_resolution_clock::now();

    // create and initialize matrices on GPU
    float *A, *B, *C;
    cudaMallocManaged(&A, N*N*sizeof(float));
    cudaMallocManaged(&B, N*N*sizeof(float));
    cudaMallocManaged(&C, N*N*sizeof(float));

    cout << "generate the matrices..." << endl;
    default_random_engine generator;
    uniform_real_distribution<float> distribution(0.0, 1.0);
    for(int i = 0; i < N*N; i++) {
        A[i] = distribution(generator);
        B[i] = distribution(generator);
    }
```

```

    C[i] = 0.0;
}

// matrix multiplication on GPU
cout << "multiply..." << endl;
dim3 block(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (N + BLOCK_SIZE - 1) / BLOCK_SIZE);
matrixMul<<<grid, block>>>(A, B, C, N);

// clean up
cout << "Cleaning up..." << endl;
cudaFree(A);
cudaFree(B);
cudaFree(C);

// print timing information
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
cout << "Time taken: " << duration.count() << " ms" << endl;

return 0;
}

```

```

# load the CUDA libraries and binaries
module load GCC CUDA

# compile our code with the nvcc compiler
nvcc -o mm-cuda mm-cuda.cu

# run the compiled program with 1x A100 GPU
srun -p gpu --gpus a100:1 ./mm-cuda

```

Result

	Single core	MPI (8 cores)	MPI (16 cores)	CUDA (GPU)
1500 x 1500 matrix	14s	4s	3s	~400ms

5000 x 5000 matrix	-	73s	39s	2s
10k x 10k matrix	-	-	-	9s

Revision #8
Created 3 July 2023 06:16:38 by Loki Ng
Updated 23 December 2023 14:21:50 by Loki Ng