Strassen's Algorithm in a Heterogeneous CPU-GPU Distributed System

A task parallel approach

Uriel Cabello [1], Amilcar Meneses [1], Liliana Barbosa [2] & José Rodríguez [1]
[1] CINVESTAV
[2] UDG.

Contact Information: Uriel Cabello
Center for Research And Advanced Studies
Department of Computer Science
Mexico City, Mexico
Email: jcabello@computacion.cs.cinvestav.mx

Abstract

In this work we review the Strassen algorithm for matrix multiplication. The major difficulties for its implementation in hybrid/heterogeneous systems are the number of programming tools required for the management of hardware and the complex pattern in memory access. The objective of this work is to take advantage of both: task and data parallelism using a framework that simplifies the implementation while improves the performance. The problem is first divided in many independent tasks mapped to computing devices, and then each task is executed using data parallelism to harness the advantages of GPUs and multicore CPUs. To verify the advantages of our approach we performed several experiments varying the size of the matrices, the number of nodes and type of computing devices. Our results show that we can achieve up to 3.4x of speedup combining CPUs and GPUs with the same application.

Introduction

The Strassen's algorithm [2] is an efficient algorithm which computes the product of two matrices A and B of size $N \times N$ using $O(N^{log_2 7})$ arithmetic operations. The major advantage of the Strassen algorithm is the reduction of the number of arithmetic operations, however one of the major disadvantages for its implementation is the complex pattern of memory accesses required on each matrix multiplication which in turn might become more expensive that the simpler $O(N^3)$ algorithm.

In spite of this remarkable restriction the algorithm can be used to perform the multiplication of large matrices which usually cannot be accomplished using a single computing device due to the lack of memory space. Hence we can exploit another important advantage of the Strassen algorithm which consists of its ability to split the workload in several independent matrix operations of size $(N/2)$ that can be executed in parallel.

In this work we provide an implementation of the algorithm pointed out before using the set of MPI extensions included in the "XSCALA" framework [1]. Those extensions are designed to facilitate the management and the transference of data among hybrid/heterogeneous computing devices.

In order to demonstrate the advantages of this appraoch we provide some results of the time required to execute the matrix multiplication under several scenarios.

Main Objectives

The main objective of this work consist of provide a scalable solution to the matrix multiplication problem that can deal with the limits of memory space in any single computing device while reducing the time required to complete the execution by using several computing devices.

The Strassen's Algorithm

The Strassen algorithm for matrix multiplication consists of the following three steps:

1. Divide the matrices A, B and C in four blocks:

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \quad (1)$$

2. Perform the following block matrix operations:

$$\begin{aligned} \mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\ \mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\ \mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\ \mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\ \mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\ \mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\ \mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2}) \end{aligned} \quad (2)$$

3. Compute the blocks of C as follows:

$$\begin{aligned} \mathbf{C}_{1,1} &:= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\ \mathbf{C}_{1,2} &:= \mathbf{M}_3 + \mathbf{M}_5 \\ \mathbf{C}_{2,1} &:= \mathbf{M}_2 + \mathbf{M}_4 \\ \mathbf{C}_{2,2} &:= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6 \end{aligned} \quad (3)$$

Load Balancing and Data Distribution

In order to harness the advantages of task parallelism and data parallelism, the execution is divided in two stages: In the first stage we create several independent tasks as is depicted in Figure 1. Each task is then mapped to a specific computing device using offline configuration files.

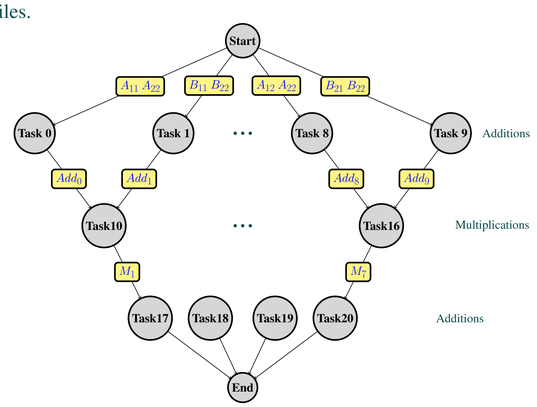

Figure 1: The directed acyclic graph for the Strassen algorithm for matrix multiplication. Tasks 0 to 9, 10 to 16, and 17 to 20 can be executed in parallel using several computing devices.

In the second stage each task executes the addition or the multiplication of the sub matrices, but in this step we use a simpler $O(N^3)$ matrix multiplication algorithm to optimize the use of shared memory on each device. The XSCALA framework uses the concept of "*tray*", to move blocks of data stored in the memory of a computing device to another regardless of their architecture (CPU or GPU) or their location (rank).

Implementation

In lines 1 and 2 of listing 1 we set the data in the memory of the computing device assigned to "TASK0". Line 3 sets the procedure or kernel on the device, and line 4 requests the execution using the globalDims and localDims parameters to establish workgroup sizes. The trays are used to indicate which blocks of memory must be used. Finally in line 7 we set a synchronization point.

Listing 1: Task based Matrix Additions

```
1 err|=OMPI_XclWriteTray(TASK0, TRAY3, SIZE/2 , A00, MPI_COMM_WORLD);
2 err|=OMPI_XclWriteTray(TASK0, TRAY4, SIZE/2 , A11, MPI_COMM_WORLD);
3 err|=OMPI_XclSetProcedure(MPI_COMM_WORLD,TASK0,srcPath,"Addition");
4 err|=OMPI_XclExecTask(MPI_COMM_SELF, TASK0, Dims, globalDims,
5                       localDims, "%T %T %T ", TRAY1, TRAY3, TRAY4);
6
7 OMPI_XclWaitAllTasks(MPI_COMM_WORLD);
```

Similarly for the execution of the seven matrix multiplications line 2 of listing 2 set up the "mtxMult" procedure of the task and line 3 requests its execution.

Listing 2: Task based Matrix Multiplications

```
1 for(taskId=10;taskId<17;taskId++){
2   err|=OMPI_XclSetProcedure(MPI_COMM_WORLD,taskId,srcPath,"mtxMult");
3   err|=OMPI_XclExecTask(MPI_COMM_SELF, taskId, Dims, globalDims,
4   localDimsMul,"%T %T %T %d %d", TRAY0, TRAY1 ,TRAY2, SIZE/2);
5 }
6 OMPI_XclWaitAllTasks(MPI_COMM_WORLD);
```

The operations in lines 2 and 3 of listing 3 enable the transference of data to compute $\mathbf{C}_{1,2}$, line 2 of listing 3 performs the copy of the memory stored in "TRAY0" of the "TASK12" to the "TRAY1" of "TASK18".

Listing 3: Inter-Task Data Transfer

```
1 int NS=SIZE/2*SIZE/2;
2 OMPI_XclSendRecv(TASK12, TRAY0, TASK18, TRAY1, NS, MPI_FLOAT);
3 OMPI_XclSendRecv(TASK14, TRAY0, TASK18, TRAY2, NS, MPI_FLOAT);
```

Similar operations are required to compute the remaining blocks.

The experimental platform consists of three nodes connected as is depicted in Figure 2.

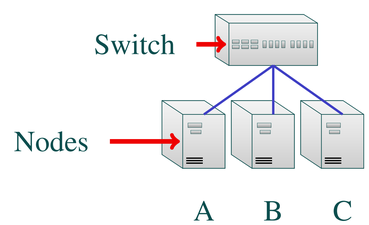

Figure 2: Architecture of the system. Each node is equipped with an Intel Core i7 processor. Node A has two NVIDIA 8400GS GPUs and nodes B and C have one NVIDIA 8400GS GPU. All nodes are connected by a cisco SG300-10P switch.

Results

In order to demonstrate the advantages of using task and data parallelism we executed several experiments varying the size of the matrix and the number of computing devices. Each experiment was performed as follows: for each matrix size we used one CPU per host, then one GPU per host, and finally the combination of all CPUs and GPUs on each host, we used one, two and the three nodes for each run. The results of the execution are summarized in Table 1.

			SIZE				
	# NODES	Node Config.	**512**	**1024**	**2048**	**4096**	**8192**
CONFIGURATION	1 NODE	CPU	1.554	1.874	4.553	29.12	221.8
		GPU	0.081	0.477	3.557	-	-
		CPU+GPU	0.685	1.004	3.533	24.39	-
	2 NODES	CPU	0.840	1.027	2.548	18.17	185.0
		GPU	0.108	0.271	2.046	15.99	-
		CPU+GPU	0.674	0.478	2.270	16.29	-
	3 NODES	CPU	1.000	0.844	2.286	15.00	164.4
		GPU	0.153	0.214	1.542	12.01	-
		CPU+GPU+GPU	1.802	0.484	1.347	8.51	93.4

Table 1: Results of the execution of Strassen's algorithm. The code has been compiled using gcc 4.6 without any compiler optimization. All results presented are measured in seconds. The "-" simbol means that the application could not be executed due to lack of memory space.

Conclusions

Taking advantage of heterogeneous computing systems is strongly related with the size of the problem being solved and with an appropriate balancing of tasks among the computing devices. Using distributed computing devices is recommended only when the size of the problem is large enough to hide the communication overhead incurred for using low bandwidth with high latency channels.

Forthcoming Research

As a future work we propose a recursive implementation of the algorithm to be able to deal with bigger matrices and to distribute more tasks to more computing devices. Another work is related with a mechanism to achieve automatic load balancing based on the capabilities of each device.

References

[1] Cabello U. Meneses A, Rodriguez J. An open mpi extension for supporting task based parallelism in heterogeneous cpu-gpu clusters. In *6th International Supercomputing Conference Mexico*, 2015. (To Appear In) Recent advances in high performance computer applications.

[2] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

Acknowledgements

The authors would like to thank to CONACyT for the grant of studies and to CINVESTAV for the infrastructure provided to the realization of this work.