PARALLEL PROGRAMMING ON GPU USING INTEL ARRAY BUILDING BLOCKS

A Thesis
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Chatura A. Ettigi
Summer 2012
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Chatura A. Ettigi:

Parallel Programming on GPU Using Intel Array Building Blocks

Joseph A. Lewis, Chair
Department of Computer Science

Kris Stewart
Department of Computer Science

Arnold Villone
Department of Mathematics and Statistics

5/7/2012
Approval Date
DEDICATION

Dedicated to Appa, Amma & Arun.
ABSTRACT OF THE THESIS

Parallel Programming on GPU Using Intel Array Building Blocks
by
Chatura A. Ettigi
Master of Science in Computer Science
San Diego State University, 2012

The goal of this project is to demonstrate Parallel Programming on a GPU using the latest Intel technology called Intel Array Building Blocks (Intel ArBB). The main aim is to describe the programming model of Intel ArBB and show effectiveness of the new technology, Intel ArBB on a GPU environment using examples. Parallel Programming is demonstrated on a NVIDIA GTX260M GPU.

We describe the evolution of GPU from graphics processor to a platform that can support numerical and parallel computation. The focus is primarily given to the Intel Array Building Block architecture, its API’s, programming constructs and their implementations on the GPU with the support of C++ Object Oriented Programming language. We discuss the ways of programming the GPU; the traditional way, where we discuss Shader programming which requires knowledge the GPU architecture and the modern way, where we use development platform like Intel ArBB (Rapid Mind) to program a GPU to execute parallelism. We show the parallelism on GPU by sample execution of Vector Product and Matrix Multiplication written in C++ (courtesy Intel corporation), with parallelism provided by Intel ArBB. Finally this thesis lays a base for more advanced GPU programming using Intel ArBB.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT ...............................................................................................................................v</td>
</tr>
<tr>
<td>LIST OF TABLES ................................................................................................................. viii</td>
</tr>
<tr>
<td>LIST OF FIGURES ................................................................................................................. ix</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS .........................................................................................................x</td>
</tr>
<tr>
<td>CHAPTER</td>
</tr>
<tr>
<td>1 INTRODUCTION .................................................................................................................1</td>
</tr>
<tr>
<td>2 BACKGROUND .................................................................................................................3</td>
</tr>
<tr>
<td>2.1 GPU Hardware and Architecture .................................................................................3</td>
</tr>
<tr>
<td>2.2 Programming the Graphics Processor Unit ...............................................................3</td>
</tr>
<tr>
<td>2.3 Challenges in GPU Programming ..............................................................................4</td>
</tr>
<tr>
<td>2.3.1 Shader Program and Parallel Programming on GPU ..............................................4</td>
</tr>
<tr>
<td>2.3.2 Vertex and Pixel Shader .........................................................................................5</td>
</tr>
<tr>
<td>2.4 GPU NVIDIA GTX 260m .........................................................................................8</td>
</tr>
<tr>
<td>2.5 Current Trends in Gpu Programming .......................................................................8</td>
</tr>
<tr>
<td>3 SAMPLE PROBLEM BACKGROUND .................................................................................10</td>
</tr>
<tr>
<td>3.1 The Sample Problem 1: Vector Product ..................................................................10</td>
</tr>
<tr>
<td>3.2 Serial Implementation of Vector Product ................................................................10</td>
</tr>
<tr>
<td>3.3 The Sample Problem 2: Matrix Multiplication .........................................................11</td>
</tr>
<tr>
<td>3.4 Serial Implementation of Matrix Multiplication ......................................................12</td>
</tr>
<tr>
<td>4 INTEL ARBB ...............................................................................................................14</td>
</tr>
<tr>
<td>4.1 Intel ArBB Inception .................................................................................................15</td>
</tr>
<tr>
<td>4.2 Working of Rapid Mind ............................................................................................15</td>
</tr>
<tr>
<td>4.3 Overcoming the GPGPU Challenges Using Intel ArBB(Rapid Mind) .................15</td>
</tr>
<tr>
<td>4.4 Working with Intel ArBB ..........................................................................................17</td>
</tr>
<tr>
<td>4.4.1 Setting Up Project on Windows OS .................................................................18</td>
</tr>
<tr>
<td>4.4.2 Sample Programming Module using Intel ArBB ...............................................18</td>
</tr>
<tr>
<td>4.5 Programming with Intel ArBB .................................................................................18</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 5.1. Table of Matrix Multiplication with Serial and Parallel Execution Time and Speed Up ................................................................. 26
LIST OF FIGURES

PAGE

Figure 2.1. An example shader program on a GPU .................................................................5
Figure 2.2. Pictorial representation of working of pixel shader and vertex shader ..........6
Figure 2.3. GPU hardware with memory and registers. CUDA memory model, the GPU NVIDIA GTX260 M follows the CUDA model ......................................................7
Figure 4.1. Rapid Mind multicore software platform ............................................................16
Figure 4.2. Integration process ..........................................................................................17
Figure 4.3. Summary of parallelism on Intel ArBB ..............................................................21
Figure 5.1. Ratio of serial execution time and parallel execution time for different size of matrices ..................................................................................................................27
Figure 5.2. GPU usage monitored using TechPowerUP GPU-Z 0.5.3 ..................................28
ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Joseph Lewis for the continuous support, for his patience, motivation, enthusiasm, an immense knowledge and for providing this opportunity to work on latest technology like Intel ArBB and its implementation on a GPU.

Besides my advisor, I would like to thank Dr. Kris Stewart and Prof. Arnold Villone for offering to serve on my defense committee.

For my family, who offered me unconditional love and support throughout the course of this thesis.
CHAPTER 1

INTRODUCTION

In the early 80’s and 90’s, processor manufacturing companies came up with the idea of a graphics card to draw lines, arcs, rectangles. Later the graphics card which came with graphics coprocessor and its own instruction set led to a whole new CPU like processor called Graphics Processor Unit (GPU). The GPU handled the display and the drawing part for the entire Graphical User Interface (GUI). With the advent of 2D, 3D technologies in computer and console games, demand for a faster GPU with better pixel display and quality increased. Over time variety of GPU’s were introduced in the market which consisted of hundreds of cores, which were easily programmable, easily installed in most desktops and laptops, readily available for the price of the CPU [1].

In early 2000’s the programmable feature of the GPU was fully put in use by computer scientists and researchers in fields such as image processing, linear algebra, statistics, 3D reconstruction and many more. They started using GPUs for computational applications and found significant improvement in the performance of solving equation, floating point calculations etc. This laid the foundation for a new field of computing called GPU computing or GPGPU or General Purpose computing on GPUs. GPU computing uses both CPU and GPU together as a computing model for processing. Intensive computations, parallel processing are done on GPU and all the sequential processing is done on CPU [1, 2].

In the early stages, all the scientific calculations where mapped to the graphical representations to get them running on the GPU, since GPU could only understand the graphic programming language. This limited the full potential utilization of a GPU. NVIDIA a major GPU manufacturing company realized this limitation and came up with the support for high level languages like C, C++ for general purpose computation .They also introduced parallel hardware architecture and high level language support for parallelism [2].

GPU performs floating point arithmetic almost 20 times faster than a CPU on today’s modern workstations [3]. Programming a GPU for everyday work and computations is a tedious task, since GPUs are designed for graphical aspect of computing. It requires either
the knowledge of the architecture, special programming language or added libraries and development platform like Intel ArBB (discussed later)

There are instances where CPU is preferred over GPU for example it’s not always true that the computation executed on a GPU is 20 times faster for the whole analysis or for the whole program [4]. There may be portions of code that will run even slower on GPU compared to CPU. There cannot be a clear comparison or a preference made for processor between CPU and GPU. We still need a CPU for all our needs. Hence it is wise to decide what part of code run on a GPU and what runs on CPU.

For this project we chose Intel ArBB to program a GPU to make use of its multi-core architecture to parallelize the sample problem. The main objective here is to use a data parallel environment to program a GPU rather than the traditional approach of using GPU specific programming language.
CHAPTER 2

BACKGROUND

In this section we will present a short background on the GPU hardware and architecture that helped in better understanding of the programming aspect of the GPU. Later we will discuss about the GPU used in this thesis, the NVIDIA GTX 260 M series. After a brief overview of the GPU used, we will discuss about the current trends in GPU programming.

2.1 GPU HARDWARE AND ARCHITECTURE

The GPU hardware is a Multiprocessor Structure with ‘N’ multiprocessors with ‘M’ cores each. In the multiprocessor architecture each core share an Instruction Unit with other core and all the processor execute the same code on different data. This parallel processing used by GPU is referred to as SIMD-Single Instruction Multiple Data.

GPU is designed for rendering pixels as an assembly called texture and process the texture into four floating point numbers, each corresponding to RGBA. The main components of a GPU are the Vector and the Pixel shaders. Vertex shader outputs the three coordinates and the Pixel shader shades them. The CPU runs the main program and sends the texture as input to the GPU when parallel processing is required [4].

A GPU is less expensive compared to other parallel processing hardware. GPU follows Moore’s Law in its hardware technology and is faster than its contemporaries [5]. The traditional use of a GPU for graphical image analysis was slowly replaced with general purpose computation (GPGPU) techniques.

2.2 PROGRAMMING THE GRAPHICS PROCESSOR UNIT

GPU’s are the main processors for the video accelerator cards. As said earlier, a GPU can be used for GPGPU parallel processing apart from the graphical usage that it is originally designed for. However, the graphical API’s will not support the GPGPU programming directly. There are several GPU programming languages available, but requires special
learning and whole new library set to program and also requires one to understand the hardware aspect. Here we discuss about Intel ArBB Development Platform (Rapid Mind) which lets user to use programming language like C++, without knowing the details of the underlying architecture.

Intel ArBB (Rapid Mind) is a compiler and runtime management system for parallel processing. Intel ArBB (Rapid Mind) supports NVIDIA GPU, AMD and Intel GPUs. It outlays a simple data parallel model of execution which is easy to understand and to use. It matches well with the parallel processing feature of a GPU. The interface lets the user use the GPU to its full power with simple C++ programs. It does not require any special extension to be used to support parallelism on a GPU. It lets user to develop new applications or extend already developed applications to run on a massively parallel environment like a GPU [4, 6, 7].

2.3 CHALLENGES IN GPU PROGRAMMING

The program paradigm of GPU is expressed in the form of graphical terms. To execute a math function or an array implementation we need to write programs that the shader can process and draw graphics to solve the function [2]. The shader is divided into two parts the Pixel Shader and the Vertex Shader. The main program that executes the linear parallel operation is the Shader program of the GPU.

2.3.1 Shader Program and Parallel Programming on GPU

A series of pixels is called the texture in a GPU. Shader program executes all the operations related to a pixel (in a texture) and sends the output as an vector of four floating point numbers, which in graphical representation corresponds to a four color code sequence RGBA. An id and a program counters are also part of the output of the shader program. The id and the program counter help in fetching the next instruction to be executed. There are columns of pixels in a texture. At a particular point two columns in both the texture present an individual and a pixel present at the same location forms the input which is a unique instruction from the individual set [7].

\[
\text{Target} = \text{source 1 op source 2} \tag{2.1}
\]
Here, source 1 comes from say texture 1, from a pixel present at the location (1, 1) in a 4x4 matrix representation of texture 1. Similarly source 2 comes from texture 2, from a pixel present at the location (1, 1) in a 4x4 matrix representation of texture 2 (Refer to Figure 2.1).

<table>
<thead>
<tr>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shader Program</td>
</tr>
<tr>
<td>Texture x= Texture x+10</td>
</tr>
<tr>
<td>Texture y= Texture y-5</td>
</tr>
</tbody>
</table>

Figure 2.1. An example shader program on a GPU.

### 2.3.2 Vertex and Pixel Shader

The shader linear parallel programming and execution lays emphasis on coherent memory access. Pixel shader programming is preferred over the vertex shaders by GPGPU applications because the pixel shaders are more in number and the output is directly fed to the memory, unlike the vertex processors that sends the output through both the rasterizer and the pixel shader sections of the GPU. However, programming the pixel shaders, developing and implementing texture maps that the shader supports to perform random-access reads from memory is a cumbersome process and need specialized data structures and language support [4] (Figure 2.2) [6].

GPU textures can be compared to arrays of modern programming languages and the shader program to be like a Kernel Program. The main program execution is done by the CPU and the texture outputs the data in form of texture for GPU to process. The output array from the GPU needs to use frame buffer object that the shader program supports to output in form of texture.

All the pixel values have to be converted to vertex positions before writing it on to the destination. All these procedures make GPU programming a difficult task. Though there are many specific GPU programming languages present, they do not program the whole system, they only aid in shader programming. Hence we use Intel ArBB Development Platform that
uses simple C++ class libraries to program a GPU and doesn’t require the use of complex GPU specific programming languages [6, 7].

To make use of the parallel feature of GPU, all the individuals are tested and run simultaneously on multiple cores. Each core has ‘M’ number of processors and each processor has separate registers. All these processors share a common memory. The important feature is the memory and its interconnection. Each multiprocessor has thread processors that run the copy of the same program and execute same instruction at the given same time (Refer to Figure 2.3) [8].

The GPU has its own memory called device memory. The memory access time of the CPU is very slow. Caches are used by CPU’s in order to reduce the memory latency and store the processed data for further use. But these caches do not solve the problem of memory bandwidth. GPU uses the concept of multithreading to solve the problem of memory bandwidth when executing huge data. When the GPU processor issues an access to
the device memory, that GPU thread goes to sleep until the memory returns the value. In the meantime, the GPU processor switches over very quickly, in hardware, to another GPU thread, and continues executing that thread. In this way, the GPU exploits program parallelism to keep busy while the slow device memory is responding.

While the device memory has long latency, the interconnection between the memory and the GPU processors supports very high bandwidth. In contrast to a CPU, the memory can keep up with the demands of data-intensive programs; instead of suffering from cache stalls, the GPU can keep busy, as long as there is enough parallelism to keep the processors busy.
2.4 GPU NVIDIA GTX 260m

The GPU used for our GPU computing purposes is the NVIDIA GTX 260m series GPU computing processor. This processor is NVIDIA’s 10th generation graphics processing unit. This is a notebook GPU, known for its High Performance. The GTX 260m processor consists of 112 cores operating at a frequency of 1375 MHz. The standard memory configuration for this processor is 1GB, with a memory bandwidth of 61 GB/sec. This is mainly a gaming Graphics card processor used for 3D gaming, with new gaming effects, character details and motion. This GPU is used in many engineering, medical, scientific and financial areas that require high computational power, superior image quality and the ones which require distributed computing environment with multiple applications running simultaneously [2, 9].

One of the key features of this processor is the advanced Scalable Processor Array architecture (SPA). This SPA architecture consists of Thread Processing Clusters which in turn consists of streaming multiprocessors which has 8 cores called thread processors making a total of 240 processor cores [10].

We perform Matrix Multiplication using Intel ArBB which uses the computational side of this processor. We use TechPower Up GPU software to monitor the usage of GPU.

2.5 CURRENT TRENDS IN GPU PROGRAMMING

NVIDIA one of the giants in GPU manufacturing industry came up with the CUDA parallel computing architecture for graphics processing [11]. This is basically a computing engine in NVIDIA GPU which is made available for many developers to use wrappers to program them. Many use C for CUDA for execution of their programs on the GPU. CUDA works only on NVIDIA GPU’s and to program CUDA one must understand the underlying architecture details.

OpenCL is another framework that is available for programming a GPU. OpenCL framework includes a separate programming language like C99 (with some modifications) which lets the programmer to program heterogeneous platforms like CPU’s, GPU’s and other processors. The main feature difference between OpenCL and CUDA is the former’s portability on the processor. Unlike CUDA which knows its underlying processor is NVIDIA
GPU, OpenCL uses its abstract memory and execution model to program and execute programs on various kinds of processors [10, 12].

There are other GPU programming frameworks like OpenMP, Microsoft’s C++ Amp and Intel’s TBB and Intel’s ArBB that help programmers to program a GPU using High Level languages like C, C++ and API’s, compilers and math libraries provided by these frameworks. One such framework that is discussed here in detail is Intel’s ArBB, the Array Building Blocks.
CHAPTER 3

SAMPLE PROBLEM BACKGROUND

In this section we discuss about Vector Multiplication and Matrix Multiplication which is our sample problem executed parallel on a GPU NVIDIA GTX 260 M using Intel ArBB. The parallelization using Intel ArBB is discussed in detail in later Chapter. Here we give a brief overview of the theoretical concepts of Vector Multiplication and Matrix Multiplication and their Serial implementation.

3.1 THE SAMPLE PROBLEM 1: VECTOR PRODUCT

The binary operation on two vectors in a three dimensional plane that will result in another vector is called “Cross Product of two vectors” or “Vector product”.

Below is the cross product on two 3D vectors:

\[
\begin{vmatrix}
|a.x| & |b.x| & a.y * b.z - a.z * b.y \\
|a.y| & |b.y| & a.z * b.x - a.x * b.z \\
|a.z| & |b.z| & a.x * b.y - a.y * b.x \\
\end{vmatrix}
\] (3.1)

Let \(A\) and \(B\) be two non collinear vectors. The cross product between \(A\) and \(B\) will result in the third vector \(C\), i.e. \(C = A \times B\). The Cartesian components or the co-efficient of the co-ordinates is expressed in terms the components of \(A\) and \(B\) [13].

Suppose \(x, y\) and \(z\) are the co-ordinates of vectors \(A\) and \(B\), then vector \(C\) is expressed as [13]:

\[
\begin{align*}
Cx &= AyBz - AzBy \\
Cy &= AzBx - AxBz \\
Cz &= AxBy - AyBx
\end{align*}
\] (3.2)

In the next section we see the C++ implementation of the above logic.

3.2 SERIAL IMPLEMENTATION OF VECTOR PRODUCT

Sample C++ code.

\[
\text{Vector Vector\_Product::operator *(const Vector &v) const}
\]
\{
    Vector result;
    result.x = (y * v.z) - (z * v.y);
    result.y = (z * v.x) - (x * v.z);
    result.z = (x * v.y) - (y * v.x);
    return result;
\}

In the above sample code, the “result” is the new vector that is obtained after the Cross Product of two vectors. The “result” will contain the coefficient of the x, y, and z co-ordinates.

Below is the Math example:

Let \( v = (2, 5, 1) \) and \( u = (-3, 2, 4) \) be 3D vectors.

The vectors can be expressed in terms of \( i, j, k \) form as: \( v = 2i + 5j + k \) and \( u = -3i + 2j + 4k \).

The cross product between vectors \( v \) and \( u \) can be mathematically expressed as a determinant of matrix by taking the components of vectors \( v \) and \( u \) along with \( i, j \) and \( k \) \[14\].

\[
v \times u = \begin{vmatrix}
i & j & k \\
2 & 5 & 1 \\
-3 & 2 & 4
\end{vmatrix} = i\begin{vmatrix}5 & 1 \\
2 & 4
\end{vmatrix} - j\begin{vmatrix}2 & 1 \\
-3 & 4
\end{vmatrix} + k\begin{vmatrix}2 & 5 \\
-3 & 2
\end{vmatrix} = 18i - 11j + 19k = (18, -11, 19) = w
\]

Here \( w \) is the new vector having the value as \((18, -11, 19)\)

### 3.3 The Sample Problem 2: Matrix Multiplication

Matrix Multiplication is a binary operation, where a matrix ‘A’ of order m by p is multiplied with the matrix ‘B’ of order p by n, resulting in a matrix ‘C’ of order m by n. In simple, it’s a way to combine to matrices resulting in a third matrix. We multiply every element across rows of matrix A with every element down column of matrix B and then add them together. In other words element in third or the resulting matrix is the sum of the products of row and column elements of matrix A and matrix B \[15\].

Despite its computational complexity, matrix multiplication is not inherently difficult to articulate. Nevertheless, we feel that this is a valuable entry point for assessing this
framework for leveraging parallelism, because it factors heavily in the application of two very important techniques: Principle Component Analysis and Singular value decomposition.

Example [15]:

\[
A = \begin{bmatrix}
  a & b \\
  c & d
\end{bmatrix}
\quad B = \begin{bmatrix}
  w & x \\
  y & z
\end{bmatrix}
\quad AB = \begin{bmatrix}
  a & b \\
  c & d
\end{bmatrix}
\begin{bmatrix}
  w & x \\
  y & z
\end{bmatrix}
= \begin{bmatrix}
  aw + by & ax + bz \\
  cw + dy & cx + dz
\end{bmatrix}\tag{3.4}
\]

In the above example, we have two matrices “A” and “B”, with elements \{a, b, c, d\} and \{w, x, y, z\}, then the matrix multiplication “AxB” is, multiply the every row element of matrix “A” with column elements of matrix “B” and then doing the summation.

Math implementation of Matrix Multiplication:

Let “A” be a 4x3 matrix and “B” be a matrix of order 3x2, then AxB is [15]:

\[
A = \begin{bmatrix}
  14 & 9 & 3 \\
  2 & 11 & 15 \\
  0 & 12 & 17 \\
  5 & 2 & 3
\end{bmatrix}
\quad B = \begin{bmatrix}
  12 & 25 \\
  9 & 10 \\
  8 & 5
\end{bmatrix}
\]

The matrix multiplication result will be a 4x 2 matrix:

\[
AB = \begin{bmatrix}
  14 & 9 & 3 \\
  2 & 11 & 15 \\
  0 & 12 & 17 \\
  5 & 2 & 3
\end{bmatrix}
\begin{bmatrix}
  12 & 25 \\
  9 & 10 \\
  8 & 5
\end{bmatrix}
= \begin{bmatrix}
  273 & 455 \\
  243 & 235 \\
  244 & 205 \\
  102 & 160
\end{bmatrix}\tag{3.5}
\]

3.4 SERIAL IMPLEMENTATION OF MATRIX MULTIPLICATION

C++ code snippet:

```cpp
void matrix_mul(float *x, float *y, int r)
{
    int a,b,c;
    float temp = 0;
```
for( a=0; a<r; a++)
{
    for(b=0; b<r; b++)
    {
        for(c=0; c<r; c++)
        {
            temp += (x[a][c] * y[c][b]);
        }
    }
}
}
CHAPTER 4

INTEL ARBB

The Intel Array Building Blocks (Intel ArBB) software is a data-parallel programming environment which provides a user C++ library interface that enables the user to write code using standard C++ and compile on available standard compilers. The user can pick the portion of the code that has more potential for parallelism and manipulate the composition and optimize its operators as suggested by Intel ArBB [16].

Inter ArBB provides various level of Abstraction with regards to Language, programming model, API and High performance Virtual Machine. Intel ArBB just mimics C++ way of control flow statements, the way types are used and operators are added. Only specific portions of C++ code needs to be rewritten with the help of runtime library and header files.

The programming model of Intel ArBB does parallelism following the sequential semantics. The new types used by Intel ArBB are created using the C++ type system. New operators for Intel ArBB are created using operator overloading or by library calls. The main entry point in the program are the call ( ) and map ( ) functions, we see them in detail later in this chapter [16].

Intel ArBB runtime has a High performance Virtual Machine (VM) with high level interface and dynamic compiler which removes the need of knowing the underlying architecture. The VM transforms the high level code into optimized and parallel machine code for target architecture. By generating optimized, parallelized code dynamically during runtime the VM can adapt to the target architecture. Intel ArBB is implemented on this VM using the C++ syntax. The VM has caches which store the function for reuse. The degrees of parallelism supported in relation to threads, the software architecture, machine details, memory allocation all are hidden.

The safety feature in Intel ArBB is taken care by making the objects non-accessible by C++ pointers. All the objects operate in a separate data space. Programs written using Intel ArBB behave same on single core and multicore platforms, but results vary.
4.1 **Intel ArBB Inception**

Intel ArBB concept was initially developed by Rapid Mind Inc. which was acquired by Intel in 2009. Through 2010 Intel sold Rapid Mind Development Platform and provided support for its customers. In May 2011, Intel integrated the Rapid Mind technology with its ongoing Intel Ct project and came up with a development platform called Intel ArBB which has the functionality of Rapid Mind with extended Intel Library set. We first give a brief overview of Rapid Mind’s working and architecture. Later in this chapter, we discuss the use of Intel ArBB, its functionality, code approach used in our project to program a GPU [16, 17].

4.2 **Working of Rapid Mind**

We will first explore and understand Rapid Mind and before we get on to the details of Intel ArBB. Rapid Mind is no new language. The whole interface is embedded in the ISO of C++ (Figure 4.1) [17].

Rapid Mind support types to create parallel programs within an existing C++ program:

- **Value**: similar to the types like float and int in C++
- **Array**: is like arrays in C or like vectors in C++
- **Program**: Contains computations, implements encapsulation as in C++.

Writing Applications and Overall Integration process on Rapid Mind (Figure 4.2) [17]:

i) Design a program to be executed on a GPU. Replace the numerical types in the program with the Rapid Mind platform types.

ii) All the numerical operations invoked by the user are recorded, captured and compiled to a program object by Rapid Mind Development Platform.

iii) The runtime environment is used to manage parallel execution of the objects that are present on a GPU.

4.3 **Overcoming the GPGPU Challenges Using Intel ArBB (Rapid Mind)**

The shader programming of the GPU for GPGPU can be avoided using Intel ArBB (Rapid Mind). We can yet program a shader to be used for graphical applications.

NVIDIA GPU’s are heavy parallel processors with 3 times more processing speed of a regular CPU to process numerical calculations and also have 5 times more Memory bandwidth compared to a CPU [2]. This processing power is utilized to program the shaders.
There are specialized programming languages available for programming shaders. GPU has its own memory different from the processor memory. All the data transfer must take place to and from this memory [6].

Intel ArBB (Rapid Mind) developer platform lets the developer use C++ programming language and a regular C++ compiler to program a GPU with fewer efforts. It provides new runtime API and also support for Windows and Linux based development. Intel ArBB (Rapid Mind) Development Platform is not a separate IDE. It provides necessary libraries, packages which the existing IDE can use for writing code to program a GPU. Single threaded programming is the main idea behind the Intel ArBB (Rapid Mind) Development Framework. For all these features, Rapid Mind was acquired by Intel in 2009. To use Rapid Mind (now Intel ArBB) one does not need to understand the complexities and
details of the underlying architecture. All these features make Rapid Mind a more sought after framework for GPU programming [16, 17].

4.4 WORKING WITH INTEL ARBB

Install the Intel ArBB software on the machine with Windows OS (used in this thesis). Check the directory where the installation is done. Set the environment variables like LIB, INCLUDE, LD_LIBRARY_PATH and CPATH [18].

Run the command – arbbvars.bat arg1 arg2

arg1 = ia32 – IA 32 architecture / intel64 – Intel64 architecture.
arg2 = Microsoft Visual Studio version and Intel TBB version (optional)
4.4.1 Setting Up Project on Windows OS

Intel ArBB supports all sorts of compilers like gcc GNU C/C++ compiler, icc Intel C/C++ compiler. Check that the install is installed in the proper directory under Program Files. There are two libraries that are available, one is arbb_dev.dll, which is the development library and arbb.dll which is the deployment library. Choose the right type of library. The arbb_dev.dll is for the application developers and arbb.dll is for redistribution. Intel ArBB requires its dynamic libraries to be installed properly. The Microsoft Visual studio also has dependency on libraries, like arbb.dll for release version and arbb_dev.dll for debug version. All the dynamic variables must be set in PATH environment variable [16].

4.4.2 Sample Programming Module using Intel ArBB

1. Include the Intel ArBB library header file arbb.hpp.
   ```
   #include <arbb.hpp>
   ```
2. Add the following statement at the front of your source code:
   ```
   using namespace arbb;
   ```
3. Move code into a Intel ArBB function by adding these lines before and after the code section;
   ```
   void foo(dense<T> in, dense<T>& out) {
       <code section>
   }
   ```
4. Initialize the dense input containers using range objects:
   ```
   dense<f32> in;
   memcpy(&in.write_only_range()[0], inPtr, length * sizeof(float));
   dense<f32> out = fill(0, length);
   ```
5. Invoke the Intel ArBB function using a call:
   ```
   call(foo)(in, out);
   ```
6. Ensure that the call has completed:
   ```
   out.read_only_range(); [16]
   ```

4.5 Programming with Intel ArBB

The parallelism in Intel ArBB is mainly based on the ‘container types’. Intel ArBB supports various types like Scalars, Arrays and Containers.

4.5.1 Scalar Types

The Scalar Types are similar to the integer and float types of C++ programs. The main difference is that they are executed by the Intel ArBB runtime library. The Scalar Types are declared in the arbb namespace.
Examples:

<table>
<thead>
<tr>
<th>Scalar Types</th>
<th>C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i8</td>
<td>char</td>
</tr>
<tr>
<td>i32</td>
<td>int</td>
</tr>
</tbody>
</table>

Scalar types are not values, they just represent values. To access the values we use `arbb::value()` function as a regular C++ type. The types in Intel ArBB are immutable and isolated from the C++ types and they cannot be modified and this is how it provides the safety feature [16].

### 4.5.2 Functions and Control Flow

Intel ArBB types are used with the C++ code block. To call such types we use `arbb::call()`, this function accepts Intel ArBB parameters and must return void. When a function having `arbb::call()` is invoked, it collects all the Intel ArBB types, optimizes, compiles and collects the computations expressed in the function.

We know that Scalars just represent values and are not values by themselves. They cannot be used directly in control flow or looping constructs, like we do in C++. There are specialized flow control and looping constructs for Intel ArBB.

Intel ArBB has macros like `_if`, `_for` and `_while` and their ending macros like `_endif` that represent the looping constructs and the control flow. Similarly while, do until, for can be expressed using macros like `_while`, `_do` and `_for`. Like in C++ the `_break`, `_continue` are the used for loop termination. All these looping constructs do not offer parallelism like “parallel_for”. The contents inside these looping constructs are parallelized in Intel ArBB [16, 19].

### 4.6 Parallelism using Intel ArBB

The parallelism feature of Intel ArBB can be expressed in high level format as follows:

1. By using data collections and vector operations using containers.

   The two types of containers are `arbb::dense` and `arbb::nested`. The dense containers are used to add, load, and manipulate operations like vectors, arrays during runtime. The operation that takes containers as parameters is called vector operations. All the computations performed on the elements of containers are independent or knows the parallelism strategies. Thus parallelism is achieved using these vector
operations. The container does element wise operations and these operations are executed in parallel.

2. By using elemental functions.

   Elemental functions are the one that act on the individual data and not on the data collections.

   `arbb::map()` operator is used to invoke the elemental function on all the operators, across the containers. The dimensions and the size of all the varying arguments passed to the map function must exactly match. [18]

   The simplest containers available are the dense containers. These containers are represented by the class template `arbb::dense`. The `arbb::dense` takes two parameters, first one is the element type and second one is the dimensions. The mapping of dense function to regular C++ memory is done using `range`. It is also serves as an interface to access values from dense containers as plain C++ data.

   All the element type operations on each element in the container operate independently. The operators (`operator()` and `operator[]`) are used to read and write the single elements of dense containers. Containers always produce new containers with result. `add_reduce` is used to reduce the result into a single element. All the elements of same row (when set to 0) and all the elements of same column (when set to 1) are reduced into a single element [16].

   `arbb::bind` is used to access the data that are already allocated using C++, from the containers. This kind of access is easier to work, but the runtime operation requires the data to be copied before the operation from the managed memory and copied back to the original destination after the operation. This movement of data impacts the performance. Hence the following Intel ArBB functions and macros are preferred. The whole parallelism process in Intel ArBB can be summarized as shown in Figure 4.3.

   1. `ARBB_CPP_ALIGN` – enables allocating global, class member or local function data.
   2. `ARBB_CPP_ALIGN_ALLOCA` – this is a macro determines the size during runtime for all the aligned data allocations on the stack.

   The aligned data is stored in the runtime memory till the container is destroyed. This reduces the work of copying the aligned data between managed memory space and runtime memory space, thereby increasing the performance during runtime. The function `arbb::call()` is used to provide a C++ function to Intel ArBB.
The parallelism on Intel ArBB can be summed up as follows:

One can encounter the following kinds of errors while working with Intel ArBB [16]:

1. **Compile-time errors**
   - These kinds of errors cause compilation errors. It usually happens when Intel ArBB tries to compile a C++ file that has syntax errors. It is caught by C++ compiler.

2. **Run-time errors**
   - An arbb::exception instance is thrown. This kind of exception is caught by C++ exception handling. All the exceptions thrown by Intel ArBB are the instances of arbb::exception.

To express the C++ code snippet in Intel ArBB we need to do the following:

- We need to have the `<arbb.hpp>` header file included in our programs,
- Appropriate Array Building Blocks objects must be created,
- The computation on these objects must be well defined, and
- To invoke a call to ArBB from C++ use the arbb::call method.
CHAPTER 5
EXPERIMENTS

In this section we discuss about Vector Multiplication and Matrix Multiplication in Parallel using Intel ArBB. With the help of code snippet we discuss the parallelism factor and its implementation on GPU. The goal here is to explain the concept of how parallelism is achieved using Intel ArBB. Finally, we compare the serial and parallel execution of the matrix multiplication.

5.1 VECTOR MULTIPLICATION/ CROSS PRODUCT PARALLEL USING INTEL ARBB

Vector Product forms the basis for numerous calculations in the field of Computational Geometry, Mechanics, Astronomy, Physics and Robotics to name a few. Parallelizing Vector Product helps in computing dense equations efficiently with less amount of time. We can parallelize Vector Product using Intel ArBB. Below is the code snippet to parallelize Vector Product [20].

```cpp
using namespace arbb;
void calc_vectorProduct(dense<F64> src1, dense<F64> src2, dense<F64> & dst1)
{
    dst1 = src1 * src2;
}
int main()
{
    double vec1[n], vec[n], result[n];
    arbb::dense<f64> Vec1;
    arbb::bind(Src1, src1, n);
    arbb::dense<f64> Vec2;
    arbb::bind(Src2, src2, n);
    arbb::dense<f64> Res;
```
arbb::bind(Res, result, n);
arbb::call(calc_vectorProduct)(Src1, Src2, Res);
}

In the above code snippet, dense<F64> vec1 is two dimensional dense container of f64 elements and vec1 is the first vector.

The container is the one that renders parallelism in Intel ArBB. The container operation on another container returns a container. The elements inside the container are executed in parallel fashion.

Intel ArBB can access the data that is already allocated using C++. There is a special function called arbb::bind binds the container data together. By default, it assumes all the data is stored in the form of pages/rows or columns. The call function by itself is not parallel. The contents inside the call function are executed in parallel. The arbb::call is used to call the calc_vectorProduct.

The arbb::bind() function described above has a container, a pointer and the number of elements as parameters. It gets the data from C++ code and passes it on to the call function, where the parallel execution is done on the elements.

dst1 = src1 * src2 - According to Intel Integrated performance primitives, multiplication is carried out on vector elements, where src1 – operand is the array of vectors and src2- operand is the single vector [16, 21].

### 5.2 Matrix Multiplication Parallel using Intel ArBB

In many numerical applications matrix multiplication plays an important part. Faster matrix multiplication means faster numerical operations.

In the serial implementation we saw that Matrix Multiplication code goes through a series of for loops. In parallel implementation the matrix multiplication calculation is within a single “for loop”, but Intel ArBB does not provide a “parallel for loop”, nor does it parallelize the for loop that is used in the code below. Parallelization is achieved and expressed in Intel ArBB through containers and/or arbb::map functions. The function invocation is a serial action, but the function call within contains parallelism, for example function containing “reduction” add_reduce is collectively executed in parallel fashion.
In matrix multiplication, we use dense containers where element wise operation happens for example: We have matrix ‘A’ and matrix ‘B’ [15],

\[
A = \begin{bmatrix}
14 & 9 & 3 \\
2 & 11 & 15 \\
0 & 12 & 17 \\
5 & 2 & 3 \\
\end{bmatrix}
\quad B = \begin{bmatrix}
12 & 25 \\
9 & 10 \\
8 & 5 \\
\end{bmatrix}
\]

(5.1)

Here matrix multiplication is sum of the product of every element in A and B [15].

\[
14 \times 12 + 9 \times 9 + 3 \times 8 = 273,
\]

\[
14 \times 25 + 9 \times 10 + 3 \times 5 = 455.
\]

\[
AB = \begin{bmatrix}
14 & 9 & 3 \\
2 & 11 & 15 \\
0 & 12 & 17 \\
5 & 2 & 3 \\
\end{bmatrix}
\begin{bmatrix}
12 & 25 \\
9 & 10 \\
8 & 5 \\
\end{bmatrix} = \begin{bmatrix}
273 & 455 \\
243 & 235 \\
244 & 205 \\
102 & 160 \\
\end{bmatrix}
\]

(5.2)

All these element wise operations are executed in parallel by dense containers i.e. every row multiplication with column item of other matrix and its subsequent addition happens in parallel.

Apart from dense containers, we use reduction operation like add_reduce and replace_col function. The add_reduce reduces the row into single element after addition, if add_reduce is set to “1” it reduces the column elements into single element after addition. The replace_col replaces the column of a container with the column of another container.

Code Snippet (Courtesy Intel Corporation):

```cpp
void parallel_matrix_multiplication(   arbb::dense<arbb::f32, 2>& a,
arbb::dense<arbb::f32, 2>& b,   arbb::dense<arbb::f32, 2>& c)
{
  using namespace arbb;

  usize x = a.num_rows();
  usize y = b.num_cols();

  _for (usize i = 0, i < y, ++i)
  {
    dense<f32, 2> multiply= a * repeat_row(b.col(i), x);
    dense<f32> column= add_reduce(multiply);
    c = replace_col(c, i, column);
  }

  return c;
}
```
Here, `arbb::dense<arbb::f32, 2>& a` is a two-dimensional dense container of `f32` elements and “a” is the matrix. The dense container does the element-wise operation.

namespace arbb – Array Building Blocks functions run in ArBB space.

`num_rows()` – and `num_cols` – is used to reading the size of the matrices.

`repeat_row` – returns a 2D container, where each row corresponds to a one-dimensional source.

`add_reduce` – all the elements in the row/column are reduced to a single element, by addition.

`replace_col` – returns a dense container, with the column replaced by another container [16].

### 5.3 RESULTS

When we run the serial matrix multiplication and ArBB implemented matrix multiplication, on different matrices of varying size we see the following execution time and respective speed gain. We can clearly see that for large matrices, GPU execution using Intel ArBB (parallelism) is much faster than the serial implementation (Table 5.1).

Figure 5.1 is the graphical representation of speed gain respect to different matrices. Speed Gain is obtained by dividing the parallel execution time with the serial execution time.

### 5.4 GPU MONITORING USING TECHPOWERUP

The GPU utilization was monitored using simple software called TechPowerUp GPU-Z version 0.5.3 (Figure 5.2). It was observed that for GPU calculations the GPU load value varied from 1%-13%, depending on the size of the problem set.
Table 5.1. Table of Matrix Multiplication with Serial and Parallel Execution Time and Speed Up

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Serial Exec Time(s)</th>
<th>Parallel Exec Time(s)</th>
<th>Speed gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>100x100</td>
<td>0.007162</td>
<td>0.003505</td>
<td>2.043366619</td>
</tr>
<tr>
<td>150x150</td>
<td>0.022541</td>
<td>0.00422</td>
<td>5.341469194</td>
</tr>
<tr>
<td>200x200</td>
<td>0.053937</td>
<td>0.005854</td>
<td>9.213700034</td>
</tr>
<tr>
<td>250x250</td>
<td>0.098172</td>
<td>0.005563</td>
<td>17.6473126</td>
</tr>
<tr>
<td>300x300</td>
<td>0.177485</td>
<td>0.007749</td>
<td>22.90424571</td>
</tr>
<tr>
<td>350x350</td>
<td>0.296376</td>
<td>0.011208</td>
<td>26.44325482</td>
</tr>
<tr>
<td>400x400</td>
<td>0.433125</td>
<td>0.010674</td>
<td>40.57757167</td>
</tr>
<tr>
<td>450x450</td>
<td>0.650167</td>
<td>0.018111</td>
<td>35.89901165</td>
</tr>
<tr>
<td>500x500</td>
<td>0.92524</td>
<td>0.019924</td>
<td>46.43846617</td>
</tr>
<tr>
<td>600x600</td>
<td>1.53384</td>
<td>0.031047</td>
<td>49.40380713</td>
</tr>
<tr>
<td>700x700</td>
<td>2.516423</td>
<td>0.043101</td>
<td>58.38432983</td>
</tr>
<tr>
<td>800x800</td>
<td>3.949067</td>
<td>0.057902</td>
<td>68.20260095</td>
</tr>
<tr>
<td>900x900</td>
<td>6.131202</td>
<td>0.098825</td>
<td>62.04100177</td>
</tr>
<tr>
<td>1000x1000</td>
<td>8.681971</td>
<td>0.108338</td>
<td>80.13781868</td>
</tr>
</tbody>
</table>
Figure 5.1. Ratio of serial execution time and parallel execution time for different size of matrices.
Figure 5.2. GPU usage monitored using TechPowerUP GPU-Z 0.5.3.
CHAPTER 6

CONCLUSION AND FUTURE WORK

There are plenty of programming models like CUDA, OpenCL, OpenMp by different industry giants that are available for GPU programming and Intel ArBB is one of the newest additions from the Intel Co-operation. The implementation and examples shown in this thesis give a brief idea of the Intel ArBB programming model, its portability feature, and its usage with high level language like C++ through the available libraries and API’s. Intel ArBB provides a platform independent framework to program a GPU. With both 32 bit and 64 bit options, it can be installed on both Linux and Windows platform and we opted for Windows for this thesis.

Every programming model has its own merits and demerits. Intel ArBB performance is competitive with all the current trends that are available. The instructions and cores are well utilized with vector and thread parallelism. A good speed up is achieved with parallel execution on a GPU using Intel ArBB. Intel ArBB can be easily integrated into existing IDE and compilers. Intel ArBB is easy to learn and to use in code that exhibits parallelism [16, 22].

Few things are beyond the scope of this thesis, like detailed view into the VM architecture of Intel ArBB, solving complex numerical problems, comparison of speed ups between Intel ArBB and other current GPU programming trends that are available in the market.

ArBB beta version was released last year and is available for developers. Application developers can use them to program Genetic Programs like Linear Genetic Programming on a GPU, to parallelize Computational Geometry algorithms using Intel ArBB on a GPU, to conduct Kinematics and Advanced Robotics experiments on a GPU, monitor the speed ups, the memory usage and memory bandwidth.
REFERENCES


