HETEROGENEOUS PARALLEL COMPUTING USING CUDA  
FOR CHEMICAL PROCESS  
Sosutha.Sa¹, Mohana.Db

¹PG Scholar, Department of Computer Science and Engineering, Coimbatore Institute of Technology, Coimbatore and 641014, India  
²Associate Professor, Department of Computer Science and Engineering, Coimbatore Institute of Technology, Coimbatore and 641014, India

Abstract

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. Using CUDA, the GPUs can be used for general purpose processing which involves parallel computation. CUDA has been used to accelerate non-graphical applications in computational biology, cryptography and other fields by an order of magnitude or more. Chemical processes need validation of their experimental data. It was found that Chemical process could become one such application where CUDA can be efficiently used. These validations of Chemical processes normally involve calculation of many coefficients. The chemical process that has been chosen for parallelizing is Heat Transfer process. This process involves calculation of coefficients for multiple iterations. As each of these iterations is independent of one another, CUDA was used to parallelize the calculation process. The execution time analysis shows that though CPU outperforms GPU when the numbers of iterations are less, when the number of iterations increase the GPU outperforms CPU greatly.

Keywords: Heterogeneous Computing, CUDA, Heat transfer application;

1. Introduction

This section provides introduction about heterogeneous parallel computing and chemical process

*Sosutha.S Email: sosutha@gmail.com
1.1 Heterogeneous Parallel Computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Heterogeneous parallel computing refers to systems that use more than one kind of processor. These are systems that gain performance not just by adding the same type of processors, but by adding dissimilar processors, usually incorporating specialized processing capabilities to handle particular tasks.

GPUs provided by NVIDIA contain same type of multiple processors which can be efficiently utilized using CUDA. These GPUs can be efficiently utilized by GPU accelerated computing. GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate scientific, engineering, and enterprise applications. GPU-accelerated computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run significantly faster.

A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential processing while a GPU consists of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. CPU aims to optimize the throughput to decrease the latency but GPU aims to increase the throughput by parallel processing rather than concentrating to reduce the latency. Figure 1 shows how CPU communicates with GPU with the help of CUDA.

![Figure 1: Communication between CPU and GPU](image)

1.2 Heat Transfer Process

1.2.1 Heat transfer

Heat transfer describes the exchange of thermal energy, between physical systems depending on the temperature and pressure, by dissipating heat. Systems which are not isolated may decrease in entropy. Most objects emit infrared thermal radiation near room temperature. The fundamental modes of heat transfer are conduction or diffusion, convection, advection and radiation.

The exchange of kinetic energy of particles through the boundary between two systems which are at different temperatures from each other or from their surroundings. Heat transfer always occurs from a region of high temperature to another region of lower temperature. Heat transfer changes the internal energy of both systems involved according to the First Law of Thermodynamics.

1.2.2 Heat transfer application

Development of high performance thermal systems has increased interest in heat transfer enhancement techniques. Heat exchangers are used to cool these high performance thermal systems. Heat exchangers are used to
transfer heat between two process streams. These heat exchangers use fluids to transfer the heat from thermal systems. Traditional heat transfer fluids like water, engine oil, ethylene glycol have limited heat transfer capability. So nano fluids are used for heat transfer.

The heat transfer process is repeated for large number of iterations and the temperature of the process fluids were recorded for each iteration. In each iteration the fluids were changed to identify the difference in the temperature.

The data obtained from the above heat transfer process includes temperature values of Nano fluids used for heat transfer. Thirteen heat transfer coefficients must be calculated for each iteration with temperature values as input. Since the input data has multiple iterations data which are independent of one another, heterogeneous parallel computing can be used effectively to calculate these coefficients for each iteration parallelly.

BACKGROUND

This section describes about different types of applications where CUDA has been used.

2.1 Comparison Study of Parallel Computing with ALU and GPU (CUDA) and MPI

Computers with high performance and speed is taking over almost all the activities in the modern world of today, reducing human effort henceforth parallelization is a feature that is exploited to provide better performance computing these days. During the execution of a C program, only sequential execution takes place resulting in wastage of time & other valuable system resources. This paper [1] presents results in parallelizing the C program such that it is executed as parallel threads by all the cores of the processor or in the case of GPU executing as threads which are passed to around 200 cores. NVIDIA's CUDA architecture provides a powerful platform for writing highly parallel programs executed by GPU. This paper is a comparison study between both ALU & GPU parallelization which results in high performance parallel computing with much more efficiency than the conventional processing.

Message Passing Interface (MPI) [2] has been the choice of high performance computing for more than a decade and it has proven its capability in delivering higher performance in parallel applications. CUDA and MPI use different programming approaches but both of them depend on the inherent parallelism of the application to be effective. This paper investigates on integration of these capabilities of both programming approaches and how superior performance can be achieved in general purpose applications.

2.2 Linear Optimization with CUDA

Optimization is an increasingly important task in many different areas, such as Finance and engineering. Typical real problems involve several hundreds of variables, and are subject to as many constraints. Several methods have been developed trying to reduce the theoretical time complexity. Nevertheless, when problems exceed reasonable dimensions they end up in a huge computational power requirement. Heterogeneous systems composed by coupling commodity CPUs and GPUs turn out to be relatively cheap, highly performing systems. Recent evolution of GPGPU (General Purpose Graphical Processing Unit) technologies gives even more powerful control over them.

Linear optimization with CUDA [3] aims at developing a parallel, GPU-supported version of an established algorithm for solving linear programming problems. The algorithm is selected among those suited for GPUs. A sequential version based on the same solving model is also developed, and experimental results are compared in terms of performance, and precision. This paper gives an idea of selecting the application for parallelizing and how to calculate the speed up obtained by parallelizing the application.

2.3 Harvesting Graphics Power for Molecular Dynamics Simulations

With the rapid increase of available computational power, more systems become tractable for simulations. Nowadays it is possible to simulate the time evolution of simple molecules over microseconds with atomistic detail on a conventional personal computer. However, for many systems, the computational power of a single processor (CPU) is not sufficient. In this case, simulations are run in parallel on many processors, to simulate hundreds of thousands of molecules over time-spans of milliseconds.
An alternative approach speeds up the simulations by using special purpose hardware. For example, in simulations of stars or charged molecules, more than 90% of the computation time is typically spent on the calculation of the gravitational or electrostatic interaction. In this paper [4], the portability of classical molecular simulation systems onto GPUs using NVIDIA’s CUDA is assessed.

2.4 Use of CUDA in V-Map Construction

In this paper [5], various ways of parallelizing the construction of the Visibility Map (V-map, a description of mutual visibility) are presented to lead the way for an implementation that achieves a speedup of 11 or more. Comparison study proposed in this paper[5] shows how the GPU process faster than CPU when the amount of data to be processed increase drastically. It provides an idea of how to analyze the performance of the parallelized CUDA program with the normal sequential program.

3. System Design

This section gives details about the system design and the underlying programming model.

3.1 Introduction to CUDA

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit. CUDA was developed with several design goals in mind:

- Provide a small set of extensions to standard programming languages, like C, that enable a straightforward implementation of parallel algorithms. With CUDA C/C++, programmers can focus on the task of parallelization of the algorithms rather than spending time on their implementation.
- Support heterogeneous computation where applications use both the CPU and GPU. Sequential portions of applications are run on the CPU, and parallel portions are offloaded to the GPU. As such, CUDA can be incrementally applied to existing applications. The CPU and GPU are treated as separate devices that have their own memory spaces. This configuration also allows simultaneous computation on the CPU and GPU without contention for memory resources.

CUDA-capable GPUs have hundreds of cores that can collectively run thousands of computing threads. These cores have shared resources including a register file and a shared memory. The on-chip shared memory allows parallel tasks running on these cores to share data without sending it over the system memory bus.

The advent of multicore CPUs and many core GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to many core GPUs with widely varying numbers of cores. At its core are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. These abstractions are to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors.

This scalable programming model allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions: from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs. A GPU is built around an array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into
blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors. This CUDA architecture can be efficiently used by writing programs in CUDA C.

3.2 Cuda Programming Model

3.2.1 Kernels

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>> execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable.

3.2.2 Thread Hierarchy

For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size $(D_x, D_y)$, the thread ID of a thread of index $(x, y)$ is $(x + y \cdot D_y)$; for a three-dimensional block of size $(D_x, D_y, D_z)$, the thread ID of a thread of index $(x, y, z)$ is $(x + y \cdot D_x + z \cdot D_x \cdot D_y)$. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks.

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which can greatly exceed. The number of threads per block and the number of blocks per grid specified in the <<<...>>> syntax can be of type int or dim3.

![Figure 2: CUDA Programming Model](image-url)
Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores. Each thread has its own memory and shared memory as well. Threads are organized as warps for processing. Number of active threads and number active blocks and the dimensions of grid and blocks vary according to the architecture of the underlying graphical processing unit.

### 3.2.3 Heterogeneous Programming

The CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program as illustrated by the figure. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.

The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory.

### 3.3 Heat Transfer Application

The application chosen to be parallelized is heat transfer process. This heat transfer application contains temperature data for different iterations for different types of fluids. Thirteen heat transfer coefficients should be calculated for each and every iteration. These calculations of coefficients are implemented in both sequential and parallel version. The algorithm for calculating the heat transfer coefficients is given.

**Algorithm**

**Input:** T1 (initial temperature), Th (change in temperature), LPM (constant value)

**Output:** T2, Tavg and other coefficients

**Steps:**
1) Allocate memory for coefficients in host (CPU)
2) Get the needed input from user
3) Allocate memory for coefficients in device (GPU)
4) Copy the data from host to device
5) Perform the calculations in parallel for every iteration device kernels

\[
T2 = T1 - th
\]

\[
Tavg = (T1 + T2/2) + 273
\]

\[
r = 983.916 + ((4.626/10)*(330 - Tavg))
\]

\[
Cp = 4184.3 + ((3.089/10)*(Tavg - 320))
\]

\[
k = 0.63971 + ((0.01147/10)*(Tavg - 320))
\]

\[
mew = 0.000489 + ((8.753*0.00001)/10)*(330 - Tavg)
\]

\[
mh = LPM*r/(1000*60)
\]

\[
m3 = (LPM)/(1000*60)
\]

\[
v = m3/0.015
\]

\[
Re = (v*0.02*r)/mew
\]

\[
Pr = (Cp*mew)/k
\]

\[
Nu = 0.023*(Re^{0.8})*Pr^{0.4}
\]

\[
hi = Nu*k / 0.02
\]

6) Copy data back to host

The algorithm shows the calculation of thirteen heat transfer coefficients. In parallel program, the
calculations are performed by individual threads in the streaming multiprocessors.

4. EXPERIMENTAL RESULTS
4.1 Experimental Environment
4.1.1. Host and Device configuration
The experiment has been run on a CPU/GPU heterogeneous system. As illustrated before, host represents the CPU used in the implementation. The CPU is a 64-bit Intel® Core™ i5-2430 M processor @ 2.4 GHz. The GPU is an NVIDIA GeForce GT 525 M. CUDA Driver Version / Runtime Version used for implementation is 6.0. GeForce GT 525M is based on Fermi micro architecture proposed by NVIDIA. Fermi is the codename for a GPU microarchitecture developed by NVIDIA as the successor to the Tesla microarchitecture. It was the primary microarchitecture used in the GeForce GeForce 400 Series and GeForce 500 Series.

Fermi Graphic Processing Units (GPUs) feature 3.0 billion transistors. It consists of

- Streaming Multiprocessor (SM): composed by 32 CUDA cores. Each SM features 32 single-precision CUDA cores, 16 load/store units, four Special Function Units (SFUs), a 64KB block of high speed on-chip memory and an interface to the L2 cache. CUDA core consist of Integer Arithmetic Logic Unit (ALU) and Floating Point Unit (FPU).
- GigaThread global scheduler: distributes thread blocks to SM thread schedulers and manages the context switches between threads during execution. The Fermi architecture uses a two-level, distributed thread scheduler.
- GigaThread Engine: The GigaThread engine schedules thread blocks to various SMs
- Dual Warp Scheduler: At the SM level, each warp scheduler distributes warps of 32 threads to its execution units. Threads are scheduled in groups of 32 threads called warps. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. The dual warp scheduler selects two warps, and issues one instruction from each warp to a group of 16 cores, 16 load/store units, or 4 SFUs.
- Host interface: connects the GPU to the CPU via a PCI-Express v2 bus (peak transfer rate of 8GB/s).

4.1.2 Implementation Details
The Operating system is Windows 7 Ultimate. The IDE used for implementation of both sequential and parallel version of application is Visual Studio 2008. Parallel version uses CUDA Run time environment embedded within the Visual Studio IDE. This CUDA run time environment is provided by the installation of CUDA driver which is available for all the GeForce GPUs in the website of NVIDIA.
C language is used for sequential version of application. CUDA C is used for parallel version of the application.

4.2 Implementation Strategy
4.2.1 Data structures used
The sequential version of the Heat transfer process was implemented in C and together with the CUDA parallel one. The data structure used to store the temperature values and heat transfer coefficients for multiple iterations is one dimensional array.
In sequential version host memory was used to hold these arrays. But in parallel version, the input data from host was transferred to device memory and calculations are performed in device and the results are copied back to host memory itself.

4.2.2 Kernels configuration
Totally ten Kernel functions work on these thirteen coefficients. Coefficients like r, Cp,k,mew are calculated in the same RCalKernel and other coefficients are calculated by individual kernels with each
kernel starting with the name same as its coefficient. RCalKernel was designed to calculate more than one coefficient since those coefficients are independent of each other.

The different kernels in the parallel program and details about the values they calculate are depicted in the table 1. Each kernel work on arrays and Single thread in each kernel work on individual temperature data of the heat exchange fluids in parallel. Since each kernel is dependent on one or more previous kernels for its input, the kernels are not executed concurrently though CUDA supports such types of concurrent execution of kernel functions.

Table 1: List of kernels

<table>
<thead>
<tr>
<th>Name of the kernel</th>
<th>Coefficients Calculated</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThCalKernel</td>
<td>Th</td>
</tr>
<tr>
<td>TavgCalKernel</td>
<td>Tavg</td>
</tr>
<tr>
<td>RCalKernel</td>
<td>r, Cp,k,mew</td>
</tr>
<tr>
<td>MhCalKernel</td>
<td>Mh</td>
</tr>
<tr>
<td>M3CalKernel</td>
<td>m3</td>
</tr>
<tr>
<td>VCalKernel</td>
<td>V</td>
</tr>
<tr>
<td>ReCalKernel</td>
<td>Re</td>
</tr>
<tr>
<td>PrCalKernel</td>
<td>Pr</td>
</tr>
<tr>
<td>NuCalKernel</td>
<td>Nu</td>
</tr>
<tr>
<td>HiCalKernel</td>
<td>Hi</td>
</tr>
</tbody>
</table>

Each kernel function use CudaMemCopy() method for transferring data to and from between host and device. Each kernel function is called by separate CUDA function.

4.3 Performance Analysis

A very important dimension in performance analysis is certainly the time. Using seconds may not be sufficient to produce good estimates. So to enlarge the precision milliseconds are used.

The OS provides developers with a library call able to retrieve the time of a system wide real time clock. The time is represented with seconds and nanoseconds since the Epoch. Unix time (aka POSIX time or Epoch time), is a system for describing instants in time, defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, not counting leap seconds. It is used widely in Unix-like and many other operating systems and file formats.

A routine GetTime64 is used to retrieve the time and find the execution time taken by the sequential program.

```c
int64 GetTimeMs64()
{
    FILETIME ft;
```
Table 2: Execution time Data

<table>
<thead>
<tr>
<th>S.No</th>
<th>Number of iterations</th>
<th>Sequential Execution time (ms)</th>
<th>Parallel Execution time (ms)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td>89.36</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>0</td>
<td>90.196</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>10000</td>
<td>0</td>
<td>95.757</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1000000</td>
<td>343</td>
<td>202.227</td>
<td>1.696</td>
</tr>
<tr>
<td>5</td>
<td>10000000</td>
<td>3245</td>
<td>1055.67</td>
<td>3.073</td>
</tr>
</tbody>
</table>

LARGE_INTEGER li;
unsigned _int64 ret;
/* Get the amount of 100 nano seconds intervals elapsed since January 1, 1601 (UTC) and copy it to a LARGE_INTEGER structure. */
GetSystemTimeAsFileTime(&ft);
li.LowPart = ft.dwLowDateTime;
li.HighPart = ft.dwHighDateTime;

ret = li.QuadPart;
ret -= 116447360000000000LL; /* Convert from file time to UNIX epoch time. */
ret /= 10000; /* From 100 nano seconds (10^-7) to 1 millisecond (10^-3) intervals */
return ret;

Figure 3: Graph comparing Sequential and parallel execution time

This routine gives the unix epoch time. The routine can be called before the calculation of coefficients and after it. The difference between the time can be used for calculating the execution time of the application. The speedup will be computed for each timed part as \( t_s / t_p \), where \( t_s \) is the sequential execution time and \( t_p \) the parallel one which is given by the following equation.
Table 2 shows the time taken by sequential program and parallel program for the calculation of thirteen heat transfer coefficients for different number of iterations of data. The execution time analysis shows that though CPU outperforms GPU when the numbers of iterations are less, when the number of iterations increase the GPU outperforms CPU greatly.

5. Conclusion and Future Work

CUDA provides a wider platform where several applications can be parallelized. Many chemical processes like heat transfer process has the potential for parallelizing and this can be done using CUDA.

The Heat transfer application involves simple calculation of heat transfer coefficients where there are no chance for concurrent kernel and shared memory GPU features, but several other applications can be identified to use these features.

Other chemical processes where parallel computing can be applied are bio reactors. Here coefficients can be calculated for different concentrations of chemicals and different flows of liquid in a bio reactor.

Control system is a chemical process where any one of the parameter like temperature or pressure is constant, other parameters are measured. This can be parallelized using Heterogeneous parallel computing. Recycling and bypass study which involves many parallel computations can also be parallelized.

REFERENCES


6. “CUDA Programming guide” by NVIDIA.