Acceleration of Distance-to-Default with GPU

Master's Thesis, TRITA-ICT-EX-2012:189

Xin Lou

System-on-Chip Design

School of Information & Communication Technology

Royal Institute of Technology

Stockholm, Sweden 2012

Abstract

Distance-to-Default(DTD), which is used to describe the default risk of a firm, acts an important role in credit research. Nowadays, since we can access a large amount of historical data, we can get a more accurate DTD result. However, this directly increases the computation time as well as the computation power. Meanwhile, Graphic Processing Unit(GPU), with its tremendous capability of parallel computing, becomes more and more popular in High Performance Computing. In addition, CUDA, a parallel computing platform and programming model that invented by Nvidia, makes GPU programming much more easier and faster. So GPU can be a proper candidate to accelerate the DTD processing if we can offload the most computation intensive part to the GPU.

In this thesis project, the author explore the existing DTD program(provided by RMI, NUS), analyze the algorithm and offload the most computation intensive part to GPU using CUDA. The platform used for testing consists of an 2.4GHz Intel i5 CPU and a Nvidia GeForce GTX460 GPU. The GPU is used as a co-processor, which is connected to the mother board with PCIe2 bus.

Acknowledgements

The first person I want to thank is Dr. Qiang Chen, my supervisor at KTH. He helps a lot on the whole procedure of the thesis project. Dr. Yajun Ha, my supervisor at NUS, also offers a lot, including office, platform etc. and I really appreciate for that. In addition, I want to thank Dr. Qiang Wu, Dr. Yi Wang, Shaobo Luo, Yongzhen Chen for their kind help during my stay at NUS.

Xin Lou, Stockholm 07/08/12

Contents

Introduction

With the great uncertainty in today's financial industry, Credit Rating Agencies are playing a more important role in guiding investigators. In order to get more accurate results, they have to process much more data then they do before and the credit rating algorithm are getting more and more complex. Distance-to-Default(DTD), which is used to describe how far a firm is away from default, is widely accepted by both academics and industry. Nowadays, DTD is one of the key indicators to guide the investigators.

1.1 Background

The NUS Risk Management Institute was established in 2006 as a research center dedicated to the area of financial risk management. [2] Measurement of Distance-to-Default is one of their main tasks and they already have a DTD prototype program running in the Matlab environment. With a large amount of historical they can access , RMI can get a more accurate DTD result then before. However, this directly increases the computation time as well as the computation power. It will take more then two days to process data from 5000 firms in the current Matlab environment. To find a way to accelerate the program is necessary for them.

Prior of this project, another group of students have tried the Field Programmable Gate Array(FPGA) acceleration. Due to the communication latency between the FPGA and the PC, they can not get any acceleration for the whole program.

1.2 Problem Statement

In the previous work that the other group have done, they have identify the most computation intensive part and converted it to VHDL. In this thesis project, the author will try the GPU acceleration of DTD program on the basis of the previous work, that means to convert the most computation intensive part to CUDA. In addition, the whole C environment should be properly set for the rest of program.

1.3 Method

As stated in the thesis registration form, this thesis project is a 24-week project, including pre-study, algorithm analysis, testing and reporting. The first stage is pre-study, which is assigned 8 weeks. In order to get a whole picture of the project, literatures about DTD should be reviewed and the CUDA programming environment should be set up. For the following 4 weeks, the author should focus on the existing matlab code, understand the DTD algorithm and identify the most computation intensive part that suitable for GPU. After that, more practical work like CUDA coding and converting the rest part of the program should be done. In the end, all parts should be assembled together and get come certain acceleration.

1.4 Delimitations

Since this is only a 24-week project, the author will only try to offload the most computation intensive part of the program to GPU using CUDA and remain the rest running on CPU.

1.5 Purpose

The purpose of this master thesis project is to accelerate the existing DTD prototype program used by RMI, NUS. In addition, this project is also part of a research project named A Power-Efficient Heterogeneous Architecture and Run-Time Manager for Data Center Servers.

1.6 Goals

There are two goals for this thesis project: 1. Port the Matlab DTD program to C+CUDA environment. 2. Get certain acceleration for the DTD program.

DTD Algorithm

The Distance-to-Default is one of the key indicators that is widely accepted by both academic and industry in the credit rating research. "The DTD computation used in the CRI system is not a standard one"[2], but an extension of the standard one which also take the financial firms into consideration. The following part of this chapter will first give an overview of DTD computation, then describe some of the key components in the DTD computation in detail.

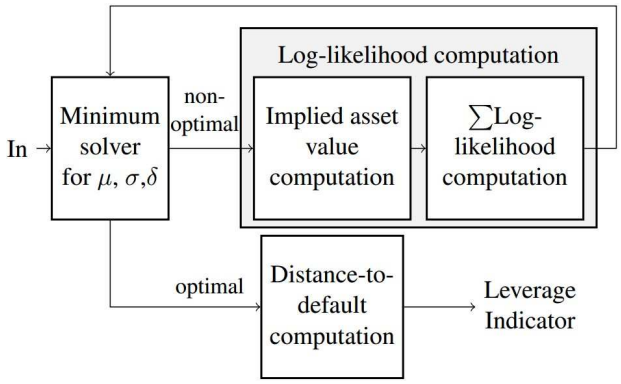

Fig. 2.1 Computation of DTD[2]

As shown in Fig.2.1, the computation of DTD mainly consists of two parts: Log-likelihood computation and minimum solver. The Log-likelihood computation itself can be divided into two parts: Imply Asset value computation and Log-likelihood computation. To compute the DTD value, three parameters- μ, σ, δ should be determined by a minimum solver where the log-likelihood function is minimized. Then the final DTD value can be computed using the three optimized parameters.

2.1 Definition of DTD

There are several theoretical models to model Distance-to-Default, and each one differs a little from others. Here in this project, according to the original DTD program provided by RMI, the Merton(1974) Model is used to Define the Distance-to-Default. On the basis of the Merton Model, DTD will be calculated as the following formula:

$$DTD_t = \frac{log(\frac{V_t}{L}) + (\mu - \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}}$$

In this formula, V_t is the present asset value of the firm, L denote the principle ceiling amount. As the formula describes, "DTD is the logarithm of the leverage ratio shifted by the expected return $(\mu - \frac{\sigma^2}{2})(T-t)$,and scaled by the volatility $\sigma\sqrt{T-t}$"[3]

2.2 Imply Asset Value

Imply Asset Value is one the most computation intensive part of the program since it should be computed for every historical data for the past 24 month. It takes the company's cap, interest rate and the company's debt as input and generate the Imply Asset Value at time T on the basis of the three optimized parameters. Two steps should be done in this phase. The first one is to determine the bound of the function using exponential-step initial search. Then the classic numerical method Newton-Raphson should be applied to calculate the responding Imply Asset Value.[2]

2.3 Log-likelihood Function

The Log-likelihood function compute the log-likelihood value for the given parameters- μ, σ, δ. The function take market capitalization, short term debt, long term debt, total liability, total asset, interest rate and number of trading days between two observations along with the corresponding calculated Imply Asset Value as input, and get the maximum Log-likelihood value for the three given parameters. The Log-likelihood value will then be passed to the Minimum Solver to see if the parameters is Minimized. This Log-likelihood computation will be iterated until the parameters are optimized.[2]

2.4 Minimum Solver

As mentioned before, the computation of DTD needs the three parameters-μ, σ, δ to be optimized. This is a constrained optimization problem and a minimum solver will be used in this phase. In this project, a C approach called MIDACO solver is used, which is a "black-box optimizer, specially developed for mixed integer nonlinear programs (MINLP)".[2]

GPU Background

As we all know, Graphic Processing Unit(GPU) is design for real-time graphics when it was first invented. However, A modern GPU is not only a powerful graphic engine, but also a general purpose computing processor which focus on parallel computing and high data bandwidth. Since the obvious diminishing of CPU speed we have seen in the past decade, more and more people turn to GPU for general purpose computing and it definitely become a hot topic since 2011. With rapid increase in both computation power and programmability, GPU could become a very promised competitor in high performance computing.

3.1 GPU Computing History[1]

The history of graphic processors is as long as the PC itself. The first graphic processor called CGA (Color Graphics Adapter) was invented by IBM in 1981. After one and half decades of development, graphic processor become more powerful and was able to support 3D acceleration on PC desktop. Then in 1999 the term Graphic Processing Unit was born when Nvidia introduced "the world's first GPU"-GeForce256. From that on, research in the fields of physics, medical imaging and so on started to take advantage of GPU to accelerate their applications. That is the beginning of GPGPU computing.

However, since the GPU was not first invented for computing, scientists had to use graphic programming language like OpenGL to program the GPU. Developer had to map their scientific applications to "graphic applications"[4]. That was really a big challenge and limited the accessibility and performance of the GPU for scientific computing.

Seeing the big market in GPU computing, Nvidia modified their GPU and made

it more programmable. In addition, a new GPU programming model CUDA was introduced by Nvidia in 2007. After that, developer can easily program the GPU using CUDA, which is just an extension of standard C.

3.2 GPU Architecture(Nvidia)

3.2.1 Overview

Nvidia is one of the biggest GPU provider in the world and its CUDA-Capable GPUs are of great performance in the field of GPU computing. Fig. 3.1 shows the basic architecture of a CUDA-capable GPU. As Fig. 3.1 illustrated, a CUDA-capable GPU is organized into a set of blocks and size of the set can be different from one generation to another. Each block is consist of two streaming multiprocessors(SMs) and the number of streaming multiprocessors generation dependent. Looking into one streaming multiprocessor, there are eight streaming processors(SPs), which share the same instruction cache and control logic. GPU used for computing always comes with a big(several giga bytes) Graphic Double Data Rate(GDDR) DRAM. The GDDR DRAM is of very high bandwidth and can be used as global memory for GPU.[5]

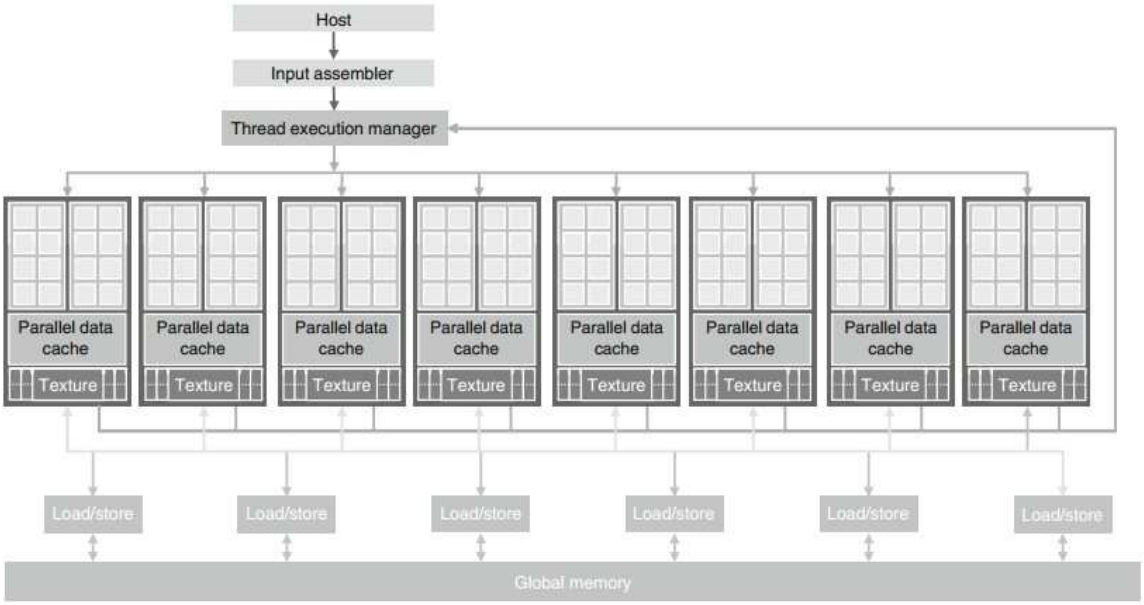

Fig. 3.1 Architecture of a CUDA-Capable GPU[5]

3.2.2 Multiprocessor Structure

A Graphic Processing Unit is consist of a number of streaming multiprocessors with several streaming processor each. Fig. 3.2 shows the structure of a multiprocessor. In Fig. 3.3 instruction unit, constant cache and texture cache are share by the streaming processors with one multiprocessor. Each streaming processor has its own register to store some frequently used data. They are small on-chip memory with very low access latency. There is also a block of memory referred as shared memory in Fig. 3.2. This is also implemented on chip with very low access latency and is designed for communication between streaming processors.

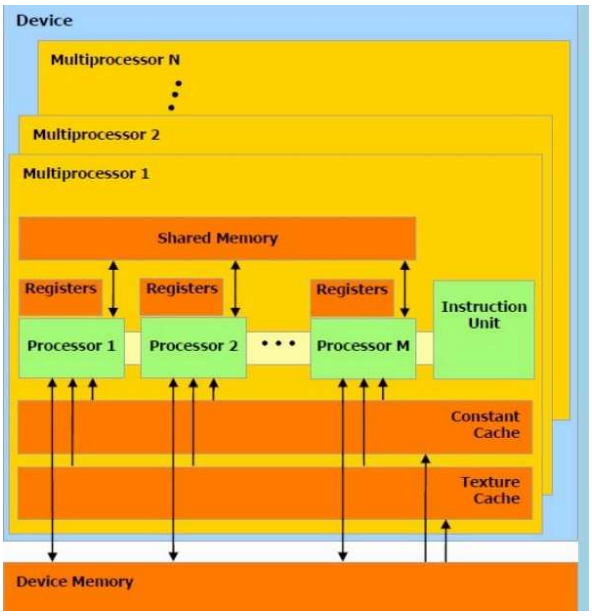

Fig. 3.3 Multiprocessor Structure[6]

3.2.3 Memory Hierarchy

As illustrated in Fig. 3.3, there are global memory, shared memory, constant cache, texture cache and register in a GPU and each of them focuses on different points. Registers are private to a streaming processor to store the most frequently used data. Constant cache is designed to cache in the constant memory. Data can be declared as constant if they will not be changed during the execution of the program. Shared memory is implemented for the purpose of communication between streaming processors. The last one global is used for communication between host CPU and GPU since GPU can not

access the CPU main memory. Data to be processed by the GPU must first copied to the global memory using corresponding API.

CUDA Programming Model

As we can see, modern computer programs are involving more and more data parallelism. However, parallel programming, especially on GPU, is really time consuming. CUDA is a programming model that design for GPU computing. CUDA computing system is consist of two parts: host and device. Usually the host part is one or several traditional CPU(s) like Intel or AMD CPUs. The device part is consist of one or more GPU(s), which is used as a co-processor. Since GPU can enjoy a lot of parallelism, CUDA devices can help to accelerate those application that have a lot of data parallelism. So parallelism is the key factor to decide if the application is suitable for a CPU-GPU system.

4.1 Program Structure

A CUDA program can usually be divided into two part: host code and device code. Host code, as the name indicated, is the code that is run on the host, which is usually a traditional CPU. It is the serial part of the program which is written in straight ANSI C. Device code, on the other hand, is the parallel part of the program which is written in ANSI C extended with CUDA keywords. A complete CUDA program is mixed source code with both host code and device code. The NVCC compiler will separate them during compilation. Then the ANSI C code will be compiled with host's standard C compiler and run on the CPU. The GPU code, also know as kernel, will be further compiled by NVCC and mapped to the GPU device.

4.1.1 CUDA Function Declaration

As stated above, a complete CUDA Program is a mixed code with both GPU and CPU code. Function declaration keywords is designed to support this kind of mix coding. As shown in Fig.4.1, functions in CUDA is declared as global, host and device. A kernel function is the function that will generate a large number threads and it is declared as global. During the compilation, the NVCC compiler will generate thousands threads for the kernel function and map them to the GPU. device is used to declare a CUDA device functions that can only be executed on GPU. In addition, a CUDA device function can only be called in a kernel function. The last keyword host is design for declaration for a host function which is run on CPU. host and device can be used to together to instruct the compiler to generate two versions for both GPU and CPU.

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

Fig. 4.1 CUDA Function Declaration[5]

4.1.2 CUDA Thread Organization

Since all thread execute the some kernel function, there should be some mechanisms to help them distinguish from others and tell them the area of data they should work on. In CUDA, all threads are organized in to a two-level hierarchy-block and grid, as shown in Fig. 4.2. A number of threads compose a block and use threadIdx to index them in a block. A grid is organized in the same way and use blockIdx to index each block in a grid. Both threadIdx and blockIdx are pre-defined variables of CUDA. In addition, there are another two pre-defined variables blockDim and gridDim, which are used to indicate the dimension block and grid by number of threads in a block and number of blocks in a grid.

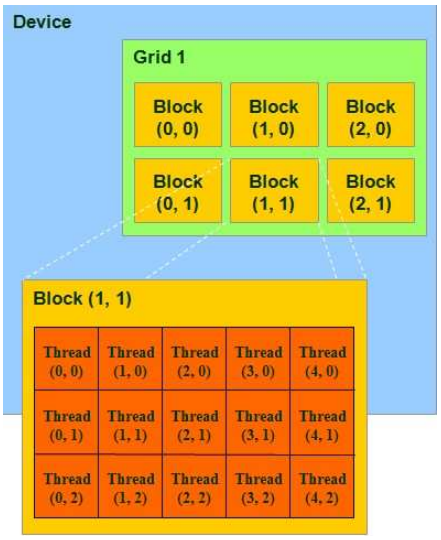

Fig. 4.2 CUDA Thread Organization[5]

4.2 CUDA Device Memory

Memory hierarchy is one of the key factors in a system. Fig. 4.3 shows the over view of the CUDA memory hierarchy. There are four kinds of memory in CUDA: global memory, constant memory, shared memory and register. As in Fig. 4.3, global memory and constant memory are used to communicate with the host device.

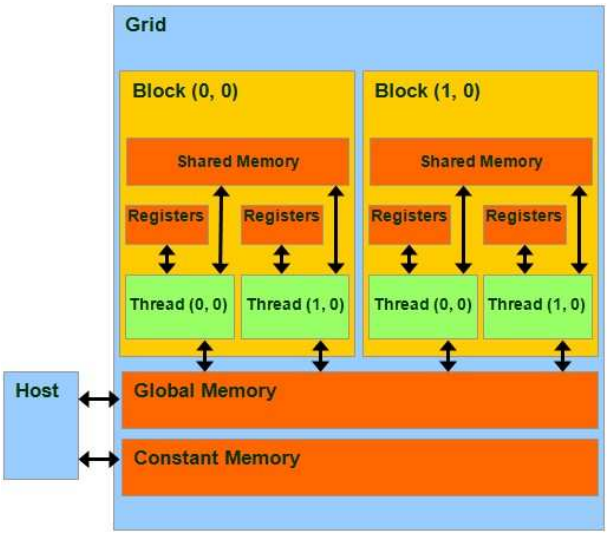

Fig. 4.3 CUDA Memory Hierarchy[5]

Global memory can be read and write in a kernel while constant memory can only be read. However access to the constant memory is much faster than the global memory since it can be cached. Shared memory is designed for the data communication for threads in a block. It is fast but very limited. All register is private to threads and is the fastest memory in CUDA. It is used for the most frequently used data in the program.Since memory access contribute a lot in the computation time of the program, developers should take advantage of different kinds of memory. The key rule is that registers and share memory should be used as much as possible and data that do not to be modified in the execution of program should be stored in the constant memory for faster access.

4.3 CUDA Code Execution

Fig. 4.4 shows the typical execution of a CUDA program. Basically there are two kinds of execution, synchronize and asynchronous.In the synchronize execution, program starts the host code, which is serially executed. When a kernel is invoked, the GPU device will take charge and a large number threads will be generated to take advantage of the parallelism. The CPU will be suspend until the GPU finish its work,and then continue executing until the end of the code. However, in the asynchronous execution, when a kernel is invoked, the CPU will continue while the kernel code is lunched on the GPU.

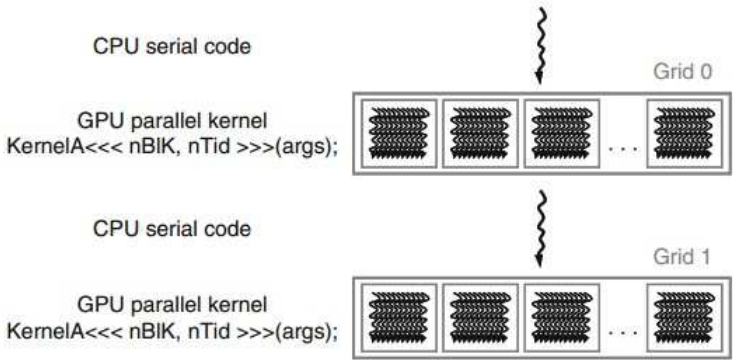

Fig. 4.4 CUDA code execution[5]

4.4 CUDA Runtime API

CUDA runtime API is a set of API functions that provide some certain kinds service, such as data transfer and so on. As shown in Fig 4.5, it is an intermediate software layer between application and CUDA drivers. Some CUDA libraries also call some CUDA runtime API. Opposite to the low level CUDA driver API, CUDA runtime API act as high level API. Each function call of the CUDA runtime API will break down to some more basic instructions that manage the low level CUDA driver API.[7]

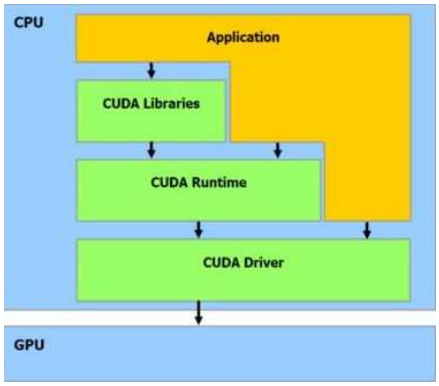

Fig. 4.5 CUDA Runtime API[7]

System Architecture

As stated before, this project is continuous work as a previous project named Acceleration of Distance-to-Default with Hardware-Software Co-Design. In this project, the author just inherit the system architecture defined by the previous and make some necessary modification. An overview of the system is illustrated in Fig.5.1. In Fig.5.1, there are mainly two parts of the system: CPU part and GPU part. The GPU card is connected to the CPU motherboard using PCIe2 16x.

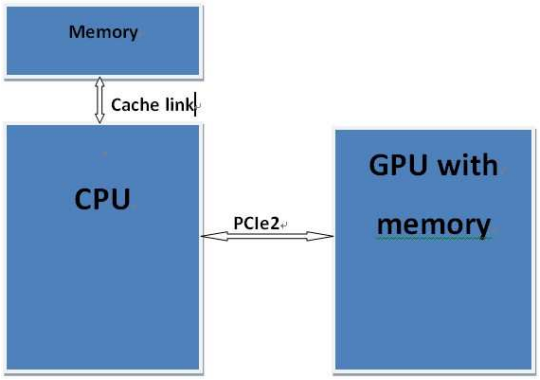

Fig. 5.1 System Overview

5.1 Matlab Implementation

The original implementation of DTD computation from RMI is in the Matlab environment. As we mentioned before, it will take as long as two days to complete the computation on a grid of several hundred computers. The flow of the matlab implementation is illustrated in Fig. 5.2.

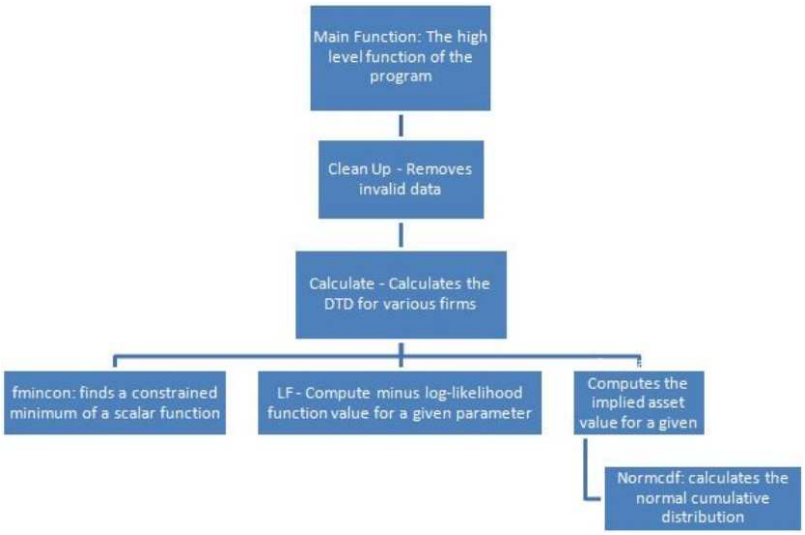

Fig. 5.2 Flow of Matlab Code[8]

5.2 Hardware Architecture

In our approach, instead of using a grid of computers, we use a Dell Precision T5500 work station as our developing environment. It has a Intel Xeon E5620 inside, which is Quad core CPU with 2.4 GHz. Along with the CPU, there are 6 GB DDR3 memory working at 1333Hz. For the GPU part, we have a Tesla C2075 which is Nvidia GPU dedicated for high performance computing. There are 448 CUDA cores working at 1.15 GHz. The double precision floating point performance is 515 Gflops and doubled the performance of 1.03 Gflops. There are 6 GB GDDR5 memory in the GPU card with a bandwidth of 144 GB/sec.

5.3 Software Architecture

5.3.1 Overview

Since we want to use GPU to accelerate the Distance-to-Default directly, we have to convert all the existing matlab program into C, identify the parallel part and offload it to the GPU. As shown in Fig. 5.3, the program will first read the data of firms from hard disk to CPU main memory. After that, the CPU code will did some preprocess for the data. Then the solver will initialize the three parameters-μ, σ, δ on the basis

of the preprocessed data and passed them to the GPU. In GPU, the Log-likelihood value will be computed using some numerical analysis methods and passed it back to the solver on CPU. This procedure will be iterated thousands of time until the three parameter-μ, σ, δ are optimized(where the Log-likelihood value is minimized). The final Distance-to-Default value will be then computed on the CPU when the three parameters are optimized.

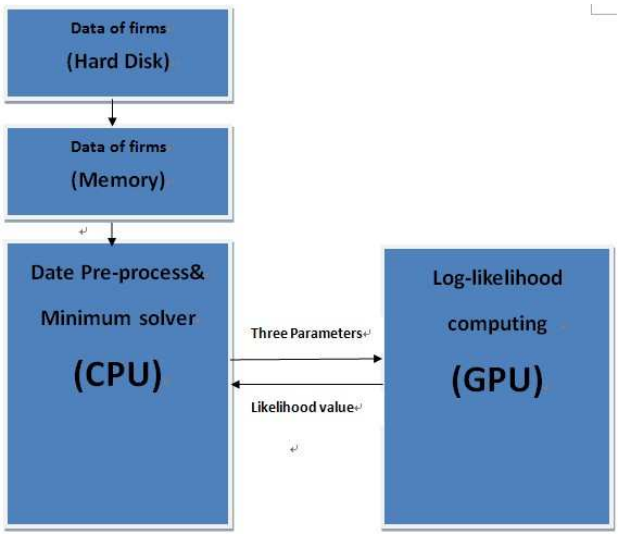

Fig. 5.3 Software Architecture

5.3.2 Asset Value Computation

In our approach, the Imply Asset Value for a firm of every trading day will be calculated in every DTD computation. The Imply Asset Value computation converted from the matlab code take four parameters: market capitalization,liability, total asset and *sigma* as input. The upper bound and low bound will be determine first, then two numerical methods-Newton-Raphson method and bisection search will be applied to compute the Imply asset value of the firm. Since this function will be called in stages of both Log-likelihood and final Distance-to-Default computation, it is declared host device function so that the compiler will generate both CPU and GPU versions for the function. The model transfer function for the Imply Asset Value is shown in Fig.5.4.

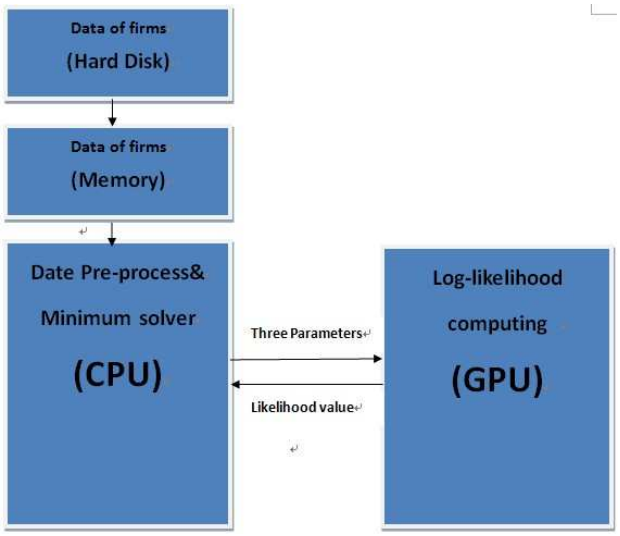

Fig. 5.4 Model transfer function for implied asset value[2]

5.3.3 Log-likelihood Computation

In this GPU accelerated version of the DTD computation, the Log-likelihood function is the most time consuming part since this function will be called thousands of time by the minimum solver to get the three optimized parameters. It will be implemented as a kernel function in this CUDA program. This function takes all the firm's seven data and three parameters as input. The liability value,which is the sum of short term debt, long term debt with weight and total liability with weight, is calculated first. Then, it will call the Imply Asset Value function to calculate the Imply Asset Value to calculate the asset value for the firm of every trading day. In the end, the three key component in the Log-likelihood computation-sum Nd1, sum logVA, sum sqterms will be determined on the basis of the two previous. The Log-likelihood value is the sum of these three component. This procedure is illustrated in Fig.5.5.

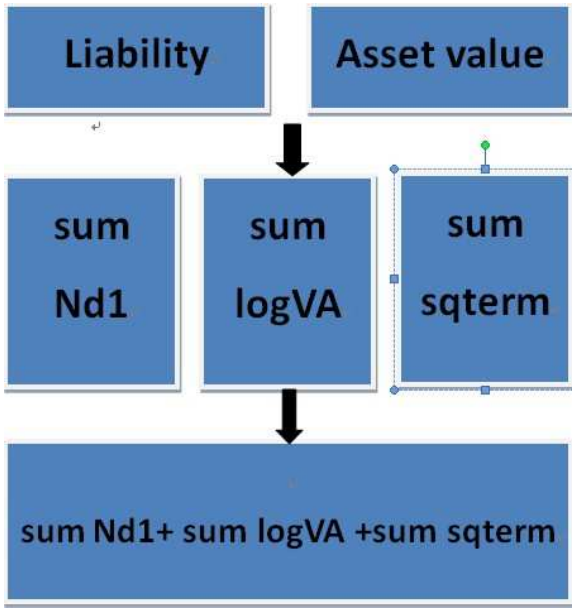

Fig. 5.5 Log-likelihood Computation

5.3.4 Midaco Solver

As mentioned before, a minimum solver is needed to determine the three optimized parameters-μ, σ, δ from a specified range. In the previous matlab implementation, a fmincon from the matlab tool box is used. In this CPU-GPU approach, a solver named MIDACO is chosen to finish the same task.

MIDACO is a commercial software from Theoretical and Computational Optimization Group,School of Mathematics, University of Birmingham. It is developed to focus especially on constrained mixed integer nonlinear programming. The algorithm MIDACO used is based on Ant Colony Optimization along with oracle penalty method. Since there are only three parameters to be optimized, we just choose a free version, which have a limitation of up to four parameter.[9] The MIDACO also support some kind of parallelism as illustrated in Fig.5.6.

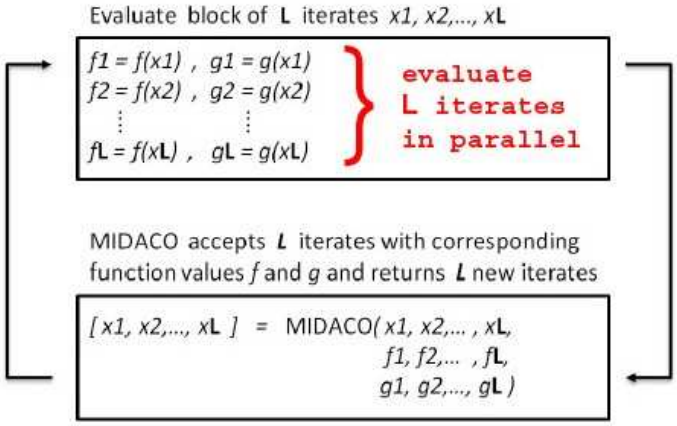

Fig. 5.6 The reverse communication loop over a block of L iterates.[9]

5.4 Performance Considerations

In this Distance-to-Default computation, there is MIDACO solver which will iterate the call of Log-likelihood function 20000 times/DTD. The solver is implemented on CPU while the Log-likelihood function is running on GPU. Therefor, the communication between GPU and CPU through a PCIe bus will be one of the bottlenecks. The other one, as mentioned before, is the memory access during the computation.

5.4.1 Communication

In this CPU GPU approach, data are frequently transferred between CPU and GPU. Before the computation of Log-likelihood value, GPU should get the firm data and corresponding parameters from CPU. While, after the computation, the Log-likelihood value should be returned for CPU to determine if the parameters are optimized. This kind of transfer will be conducted 20000 time for one DTD calculation. In order to reduce the communication time, two aspects are considered. The first one is pre-fetch. Data of the firms are divided into blocks, we transfer one block of data that covers several companies to the constant memory of the GPU which is cached. The access will be much faster if the data is in the cache. The second way be considered is memory map. CPU memory and GPU memory are mapped to each other using cudaHostAlloc and cudaHostGetDevicePointer, which enables the GPU to write back the Log-likelihood value directly.

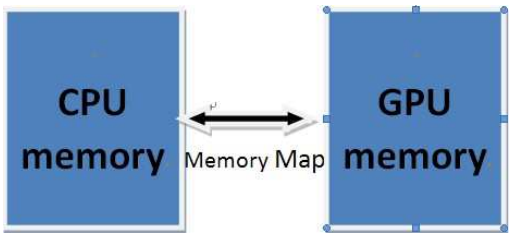

Fig. 5.7 Memory Map

5.4.2 Memory Access

The choice of memory in a CUDA is always a tradeoff. Resisters and shared memory are fast but very limited while global memory is huge but very slow, typically hundreds of clock cycles. Constant memory is a compromising one, but it need the data to be unmodified during the execution. In this approach, since the observed data of firms will not be changed, it is put in the constant memory. Imply Asset Value computed by the Imply Asset Value function will be stored in the shared memory to be shared by all the threads in a block. Other intermediate variables is declare as registers to have a faster access.

Data Type	Memory Type
Firm Data	Global memory
Imply Asset Value, Sub_sum	Shared memory
Other intermediate variables	Register

Fig. 5.8 Memory Allocation

Evaluation

According to the system architecture discussed above, experiment are conducted in order to test the performance of the GPU acceleration. Since the execution time of the program is depend on the iteration times of the solver, the performance of the matlab code and GPU code is not comparable. So we develop a pure C version of the program so that we have a reference. The pure C code is executed on the Intel Xeon E5620. The GPU version is based on a Tesla C2075 which provide the test environment of the CUDA program. The data used for test is 6175 observation of 25 companies.

6.1 Total Acceleration

For a convenient comparison, we took the DTD result of one firm as am example. As shown in Fig. 6.1, The acceleration for the whole program is about 1.8. Since at least 250 threads for the kernel is launched, the speed up is much less than expected. Looking into the internal part of the program, we find the reasons.

6.1.1 Kennel execution Analysis

In this project, the Log-likelihood function is the most computation intensive part and is implemented as device code running GPU. By measuring the execution time of different part of kernel, we find the reason for the low speedup. As shown in Fig.6.1, the computation of Imply Asset Value(call the device function in the kernel) takes an average time of 50.2Kcycles and the computation take 122.6Kcycles. However the overhead to launch a kernel is 318Kcycles, which means overhead contribute most of the execution

time in this program. In addition, the execution time of the solver that run on the CPU is neglect.

Parts	Execution time(Kcycle)
Imply Asset Value	50.2
Log-likelihood	122.6
Overhead	318

Fig. 6.1 Kernel Execution.

Discussion and Future Work

In this project, we try to accelerate the DTD computation by directly porting the original matlab code into CUDA without any modification in the algorithm. We get a certain speed up of 1.8, which is much less than expected. The main reason for the low speed up , as we discuss in the last chapter, is the overhead of of the kernel. It contributes too much in the kernel computation. In other words, our kernel is too light. Another reason is that the number of threads launched is not big enough to get a tremendous speed up. In our project, there are only 250-720 kernel launched for one time,which is limited by the current algorithm. While in a typical CUDA program, the number could be thousands.

In order to get more speed up, here are some advise: (1) Increase the number of threads launch. Since parallelism is the key factor in GPU computing, we should try to find parallelism as much as possible. In this project, this should be done by modifying the current algorithm. Another way could be CPU multi-thread since more solvers means more threads. (2) Try to put more computation in one thread. As discussed above, the kernel in this project is so light that the overhead of the kernel contribute the major part of the execution time. This can be settled by put more computation in one kernel. For this project, we can try to compute more than one Log-likelihood value in kernel.

Bibliography

[1] R. Borgo, K. Brodlie, State of the Art Report on GPU Visualization .

[2] I. Allugundu, P. Puranik, Y. P. Lo, A. Kumar, Acceleration of Distance-to-Default with Hardware-Software Co-Design .

[3] J.-C. Duan, Measuring distance-to-default for financial and non-financial firms.

[4] History of GPU Computing.
URL http://hpce.iitm.ac.in/website/gpu_history.html

[5] Daivd.B.Kirk, W. mei Wu.Hwu, Programming Massively Parallel Processors, Elsevier, 2010.

[6] A. Lippert, NVIDIA GPU Architecture for General Purpose Computing .

[7] The CUDA APIs.
URL http://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954-6.html

[8] I. Allugundu, Acceleration of Distance-to-Default with Hardware-Software Co-Design .

[9] M. Schlueter, Mixed Integer Distributed Ant Colony Optimization, University of Birmingham.
URL http://www.midaco-solver.com/parallel.html

[10] J. Brody, P. Yager, R. Goldstein, R. Austin, Biotechnology at low Reynolds numbers, Biophysical Journal 71 (6) (1996) 3430–3441.
URL http://linkinghub.elsevier.com/retrieve/pii/S0006349596795383