Standard front page

IT University of Copenhagen

Course exam with submission

Class code:

____

____

Name of course:

____

Course manager:

____

____

Course e-portfolio:

____

Thesis/project exam with project agreement

Supervisor:

Peter Sestoft

____

____

Thesis or project title:

CUDA GPGPU programming using F#

Alea.cuBase and Actulus CalcSpec for

parallelized pension reserve estimation

Name(s):	Birthdate and year:	ITU-mail:
1. Nicolai Skovvart	23/12-88	nbsk@
2.		@
3.		@
4.		@
5.		@
6.		@
7.		@

CUDA GPGPU programming using F# Alea.cuBase and Actulus CalcSpec for parallelized pension reserve estimation

Nicolai Bo Skovvart nbsk@itu.dk
Superviser: Peter Sestoft

IT University of Copenhagen

Abstract. Parallelization can provide great performance increases for large-scale independent numerical computations, for example in the field of pension reserve estimation. NVidia's CUDA platform allows for utilization of the parallelization power of the Graphic Processing Unit (GPU), but development is typically done using CUDA C based on the somewhat archaic C/C++ languages. Much of industry would also benefit from being able to utilize the power of the GPU on more modern platforms such as Microsoft's .NET.
This thesis investigates the benefits of transforming a single-threaded pension reserve estimator to utilize parallelization on the GPU on the CUDA platform. A base-line implementation is done in the somewhat archaic CUDA C language and is then ported to the more modern F# language with the aid of the language integrated compiler Alea.cuBase and the performance implications of this is explored. The F# solution is further enhanced with the capability of parsing pension plans written in Actulus CalcSpec that are automatically transformed into F# Alea.cuBase GPU kernels. The performance implications of this is also explored.
The thesis concludes that software such as pension reserve estimators can greatly benefit from parallelization on the GPU and that the benefits of developing on a modern platform such as F# far outweigh the costs. The parallelized version is able to compute up to xx times more computations compared to the single-threaded version allowing for estimations covering vastly more pension plans as well as different customers.

Keywords: GPU Parallelization, Pension Reserve Estimation, CUDA C, F# Alea.cuBase, Actulus CalcSpec

Table of Contents

List of Figures

1 Introduction

As Central Processing Unit (CPU) clock-rate growth have mostly stagnated in the recent century due partially to power inefficiency ((**QUOTE -** `http://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled` **or** `http://en.wikipedia.org/wiki/Frequency_scaling`**?**), parallelization is seen as one of primary techniques for speeding up applications.

While parallelization can be utilized by multi-core CPUs, General Purpose Graphics Processing Units (GPGPU's) in particular are used when computations on large amounts of data is needed, despite originating from the Graphics Processing Unit (GPU) designed for performing calculations in relation to computer graphics.

Parallelization is especially useful for large amounts of independent calculations that can be found in financial fields such as banking or pension.

GPGPU programming is often done on the Compute Unified Device Architecture (CUDA) platform by NVIDIA [**?**]. The CUDA platform supports multiple languages, for example as Python and Fortran, it is primarily used by NVIDIA's CUDA C (**QUOTE**) which is based on the somewhat archaic C/C++ languages.

In this thesis a single-threaded C# pension reserve estimator will be be translated to the CUDA platform and parallelized and performance gains will be explored and analyzed. First a base-line implementation will be done in CUDA C and later a solution will be implemented on the more modern .NET platform in the F# language using the language-integrated compiler Alea.cuBase. This is to allow for more rapid development utilizing GPGPU parallelization as well as allowing it to be incorporated in a .NET work-flow.

1.1 Reading guide

Subjects will be described in various levels of detail depending on their relevance to the main topics of the thesis. This means that certain languages and paradigms will be described in less detail than strictly required background information and solution descriptions.

The thesis is structured in the following chapters.

The first chapter will introduce the thesis, including the motivation, the scope and approach/method used.

The second chapter will cover a lot of the background knowledge including the math used, parallelization in general and how the GPU facilitates it, the CUDA platform and the CUDA C language, the F# language and the language integrated compiler Alea.cuBase.

The third chapter will cover descriptions of the various implemented solutions, starting with the initial single-threaded C# solution, followed by the "native" CUDA C implementation and finally the solution in F#/Alea.cuBase.

The fourth chapter will describe how pension plans written in Actulus CalcSpec are parsed and transformed to F#/Alea.cuBase kernels and the performance ramifications of this.

The fifth chapter will discuss how various alterations to the project affected performance and discuss their viability. These alterations include adding customer and pension plan variables, various memory types and the order of execution and various optimization strategies are attempted.

The sixth chapter will discuss similarities and differences to related work. Finally, the seventh chapter will conclude the thesis.

2 Background

The background section will cover four topics. First the mathematics employed in this project will be covered. This consists of Thiele's differential equation and the fourth-order Runge-Kutta method for solving differential equations. This is followed by general information on parallelization and why the GPU is particularly effective at it. After this the CUDA platform, CUDA C and F# with Alea.cuBase will be discussed.

2.1 Thiele's differential equation and the Runge-Kutta Method

The goal of the computations done in this thesis is to estimate the amount of money (reserve) required to be able to fulfil the obligations of a given insurance plan. Thiele's differential equation[**?**] called "the fundament of modern life insurance mathematics"[**?**] is a basic tool for determining conditional expected values in intensity-driven Markov processes. It can be used to express a variety of life insurance products described by multi-state Markov processes. The equation is expressed below in equation 1. It consists of the following parts:

- V_t is the reserve at time t
- π_t is the premium paid at time t
- b_t is the benefit paid by the insurer at time t
- μ_{x+t} is the mortality intensity at time t with a person of age age x at the time of signing the contract
- r_t is interest-rate at time t

$$\frac{d}{dt}V_t = \pi_t - b_t\mu_{x+t} + (r_t + \mu_{x+t})V_t \tag{1}$$

With a differential equation such as this, we need a way to solve it. The Runge-Kutta method[**?**] is a method for integrating ordinary differential equations by using a trial step at the midpoint of an interval to cancel out lower-order error terms. The equation can be seen below in equation 2.

$$\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\
k_3 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \\
k_4 &= hf(x_n + h, y_n + k_3) \\
y_{n+1} &= y_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + O(h^5)
\end{aligned} \tag{2}$$

2.2 Parallelization and the GPU

Parallelization is the act of taking one task and splitting it into smaller tasks that run concurrently (”in parallel”). This can be done on multi-core CPU’s utilizing constructs such as processes and threads, but it can especially be utilized by GPU’s that were designed to run operations in parallel for graphics processing.

This is because unlike the CPU which is specialized for data-caching and flow control, the GPU has more transistors dedicated to data processing as illustrated by figure 1.

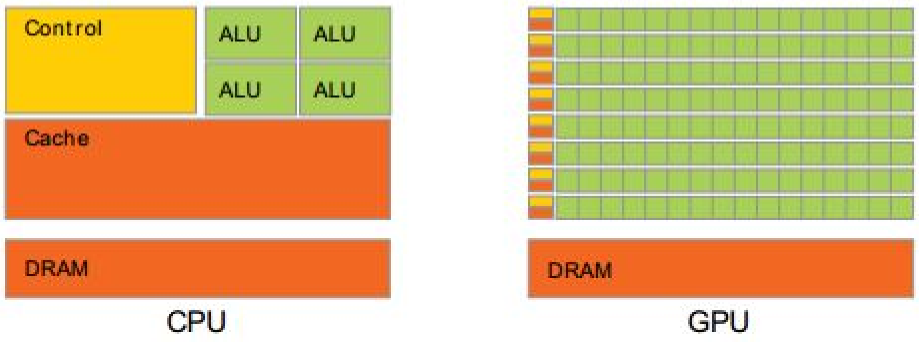

Fig. 1: The GPU devotes more transistors to data processing[**?**]

Talk about streaming multi-processors. Talk about blocks and threads (+dimensions). Talk about memory types (shared, global, local, host, etc.)

2.3 CUDA and CUDA C

CUDA is NVIDIAs parallel computing platform and programming model for harnessing the power of the GPU. The CUDA platform consists of the CUDA C and C++ languages (referred to as CUDA C from this point), based on C and C++ with some extension and limitations. It also includes parallel computing extensions for other languages such as Fortran and Python. It also includes the

CUDA Toolkit which includes a compiler, math libraries and tools for debugging and optimizing performance as well as guides, user manuals, API references and other documentation.

CUDA C is mostly C with added declaration specifiers for methods and variables. The most essential declaration specifier is the **__global__** specifier which declares a method to be a **kernel**. Kernels are void-methods that will be executed on the GPU. Code sample 1.1 show a simple kernel performing vector addition of the vectors of size N.

```
__global__ void Add(float* a, float* b, float* result){
        //unique thread id within the kernel
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        // (blockIdx always 0 in this example)
        result[i] = a[i] + b[i];
}

int main(){
        ...//Initiate memory on host and device
        Add<<<1, N>>>(a, b, result);
        ...//Copy back results, free memory as appropriate
}
```

Code Sample 1.1: CUDA C addition kernel

Another thing to note is the launch-parameters of the kernel which specify the amount of blocks and the amount of threads per block. A block can be organized into either one-dimensional, two-dimensional or three-dimensional grids. This is convenient in the case of working with for example matrices, but is not utilized in this thesis and will not be covered in further detail.

CUDA C also has other declaration specifiers. **__device__** specifies that a method will only be compiled for the GPU (which can be used by kernels), **__host__** specifies that a method will be available for the CPU. These two specifiers can also be used together for compilation to both platforms. Variables have the **__device__** qualifier as well as **__constant__** to indicate it should be stored in constant memory or **__shared__** for shared memory.

As it is based on C, dynamic memory must be allocated using malloc and subsequently de-allocated by free. This operation also has to be done for the device using the equivalent cudaMalloc and cudaFree. The tediousness of this is one of the many benefits from using a more modern language, but you do lose some control that could potentially lead to loss in performance.

2.4 F# and Alea.cuBase

F#[**?**] is an open source, cross-platform function programming language originated from Microsoft that runs on the .NET platform. Alea.cuBase by QuantAlea[**?**] is a commercial language integrated compiler that allows for CUDA development on the F# platform and in extension all .NET languages. Other than allowing for rapid development in a modern language, it features dynamic kernel compilation.

It utilizes F#'s meta-programming support Quotation Expressions[?] and is based around CUDA program templates.

```
let Square = cuda {
  let! kernel = <@ fun (a:deviceptr<int>) (result:deviceptr<int>) ->
      let i = blockIdx.x * blockDim.x + threadIdx.x
      result.[i] <- a.[i] * a.[i] @> |> Compiler.DefineKernel
  return Entry(fun program ->
    let worker = program.Worker
    let kernel = program.Apply kernel
    //return host-execution method
    fun (a:int[]) ->
      use a = worker.Malloc(a)
      use result = worker.Malloc(Array.zeroCreate a.Length)
      kernel.Launch (LaunchParam(1, a.Length)) a.Ptr result.Ptr
      result.Gather()) //return result
}

[<EntryPoint>]
let main argv =
  let a = [| for i in 1 .. 10 -> i |]
  use program = Square |> Compiler.load Worker.Default
  program.Run a |> Array.iter (fun e -> printfn "%d" e)
  0
```

Code Sample 1.2: Alea.cuBase square kernel

3 Solutions

3.1 Initial C# Solution

Describe original (C#) solution

3.2 CUDA C Solution

Describe implemented CUDA solution

3.3 F# Alea.cuBase Solution

Describe implemented F# Alea.cuBase solution

3.4 User and pension plan parameters

Talk about how running the same thing over and over again is not a very good idea, and how user and pension plan parameters are handled and their effect on performance.

4 Actulus CalcSpec parsing and code generation

Describe how Actulus CalcSpec is translated to F# and its effect on performance

5 Performance explorations

Describe various optimization strategies and their effect on the project. Lots of graphs and stuff.

6 Related Work

Related work, include Dahl+Harrington for sure

7 Conclusion

Conclude

References

A Test

Hey