Reservoir Simulation of Foam Flow using a Kepler GPU

Ismail Yahya Afiff

Theoretical GFLOP/s

Tesla K40
World’s Fastest Accelerator for Supercomputing and Big Data Analytics
Reservoir Simulation of Foam Flow Using a Kepler GPU

Table of Contents

ABSTRACT 2

SECTION 1. INTRODUCTION 3

SECTION 2. KEPLER GENERATION GPUS 3

SECTION 3. IMPLEMENTATION IN CUDA GPU 4

3.1. MATRIX ASSEMBLY, FLUX INTERFACE CALCULATION AND SATURATION UPDATE 5
3.1.1. Maximizing Exposed Parallelism 6
3.1.2. Coalescing Global Memory Access 8
3.1.3. Reducing Redundant Global Memory Access by Using Explicit Shared Memory 9
3.1.4. Using Pre-computed Table Compute Intensive Function 11
3.1.5. Using Various CUDA Read-Only Memories 11

3.2. SOLVER FOR LINEAR SYSTEM OF PRESSURE EQUATIONS 12
3.2.1. Solver and Preconditioner 12
3.2.2. Sparse Matrix Format 12

SECTION 4. RESULT AND ANALYSIS 13

Analysis 15

SECTION 5. CONCLUSIONS 16

SECTION 6. REFERENCES 16
Abstract

In recent years, along with the higher GPU's computational speed and memory bandwidth compared to those of CPUs, GPU-accelerated reservoir simulation has been studied quite extensively. The results so far have shown that Fermi generation GPUs could accelerate IMPES (implicit pressure explicit saturation) reservoir simulation considerably.

Along with several new features, the current generation Kepler GPU has been improved from its previous generation by having much higher FLOPS and memory bandwidth. However, major changes in Kepler, such as the removal of automatic L1 global memory caching and the requirement of instruction level parallelism, make additional optimizations essential to obtain close-to-peak performance.

On the application side, researchers have found that foam can be used to improve gas injection by addressing several causes of poor gas sweep efficiency. However, foam simulation is hampered by long simulation time because of its large fractional flow slope.

In this paper, it will be discussed how to implement efficient IMPES reservoir simulation on current generation Kepler GPUs, and how to apply it to foam simulation. The IMPES code is optimized by maximizing exposed parallelism (thread and instruction level parallelism - TLP and ILP), coalescing global memory access, reducing redundant global memory access by explicitly using shared memory via warp specialization while avoiding memory bank conflict, using 1D texture memory as a pre-computed table and using various forms of GPU read-only memory. Furthermore, since reservoir simulation components such as sparse matrix formats, preconditioners and solvers that work excellent on CPUs might not be efficient on GPUs, the components are chosen so that they not only just work efficiently for foam simulation but also for foam simulation on GPUs.

For the example considered in this report, and using a GTX Titan Black GPU, speed-ups up to 129 times in the saturation update and matrix assembly can be obtained compared to a parallel implementation on an Intel quad Core i7-4770k. For the pressure solver part, the GPU implementation is up to 39 x faster compared to the CPU implementation. The maximum solver speedup is achieved for large models (with more than 6 million grid cells), whereas for smaller models (a million cells or less), the speedup is reduced because GPU-CPU data transfer latency might still be dominant and small data fits in the CPU cache. Overall, the use of a GPU makes large-scale foam simulations (with six million grid cells or more) possible to be completed in days, whereas it is predicted that it will take months to complete the same simulation using a CPU.
Section 1. Introduction

In the petroleum industry, implicit pressure, explicit saturation (IMPES) is a popular numerical scheme in reservoir simulation because it avoids solving large linear systems of equation simultaneously as in the fully implicit method; furthermore, it is also very powerful for solving two-phase flow, particularly for incompressible or slightly compressible fluids (Chen, Huan and Ma 2006). In recent years, along with the higher graphical processing unit’s (GPU’s) computation speed and memory bandwidth compared to those of central processing units (CPUs), GPU-accelerated reservoir simulation has been studied quite extensively. A successful GPU-accelerated IMPES simulation scheme has been implemented by Marathon oil company assisted by Stone Ridge Technology. Using a Tesla C2075 GPU they obtain a 110 x speedup of the saturation update and a 10 x speedup of the pressure solver compared to an Intel Xeon X5560s (Esler, et al. 2011). Although the saturation update is compared with a un-optimized single core CPU implementation, the results prove that IMPES reservoir simulation could be accelerated significantly by using GPUs.

Kepler GPUs such as the Gefore Titan Black, have a much higher compute ability ($5795 \times 10^9$ floating point operations per second ($5795$ Gflops)) and memory bandwidth (336 Gb/s) compared to Fermi-based GPUs such as the Tesla C2075 (1030 Gflops and 144 Gb/s). Therefore, a significant speedup can be expected not only for compute-bound programs but also for memory bandwidth-bound programs. Besides that, Kepler also brings several new features such as dynamic parallelism, bindless texture and read-only data cache which are expected to improve the speedup and make the implementation easier.

However, there are several major changes in Kepler which result in a lower performance ratio on non-optimized code compared to Fermi (Maruyama and Aoki 2014). The major changes compared to Fermi based Tesla GPUs are the removal of automatic L1 global memory caching and the requirement of instruction level parallelism to obtain single-precision peak performance. On top of that, optimizations that work well on Fermi might not work well on Kepler because of several minor architectural changes.

On the application side, foam EOR has huge potential to improve oil recovery (Rossen 1996). However, since foam has a large fractional flow slope, the explicit saturation simulation requires a small time step size to meet the stability requirements. Hence, IMPES foam simulations requires a lot of time to complete. Therefore, it is beneficial to know if the use of GPUs could accelerate foam simulation.

Section 2. Kepler Generation GPUs

Kepler is a GPU microarchitecture successor to a Fermi microarchitecture. Kepler based GPU brings improvement not only in computational speed but also in memory bandwidth. Besides that, Kepler GPUs offer a huge leap in power efficiency, delivering up to 3 x the performance per Watt compared to Fermi GPUs (Nvidia 2012). To achieve those huge improvement, there are several major
changes introduced in Kepler. The changes not only enable better theoretical performance than its predecessor, but also better utilization of its computational potential. However, those changes also make extra optimizations necessary to obtain close-to-peak performance on Kepler compared to Fermi (Maruyama and Aoki 2014).

Kepler GPUs have much more cores per streaming multiprocessor (SM) than their Fermi predecessors, and therefore, to fill an arithmetic pipeline, more parallelism per SM is required in Kepler GPUs. This can be achieved via either an increased number of active warps of threads, or increased instruction-level parallelism (ILP) or some combination thereof (Nvidia 2014). However, the most important thing is that compared to the Fermi GF100 chip, Kepler GPUs implement in-order dual issue superscalars in their architecture by having more cores than the number of threads than can be dispatch in a cycle. Consequently, some degree of ILP in conjunction with TLP is required by Kepler GPUs in order to approach peak single-precision performance (Nvidia 2014).

In Fermi GPUs, global memory loads and writes are cached in L1 and L2. However, this behavior is changed in Kepler GPUs and also in forthcoming Maxwell GPUs. In Kepler GPUs, L1 is used only for local memory accesses such as register spills and stack data. This change makes that programs that rely on a high bandwidth, low latency, on-chip, automatic L1 cache should be re-optimized to use the read-only data cache or shared memory explicitly to obtain better performance.

In addition, Kepler GPUs also bring new features such as dynamic parallelism, improvement in texture memory usage such as bindless texture and read-only data cache, and many other nice features. Dynamic parallelism enables GPUs to launch new GPU kernels without the need of a CPU. In reservoir simulation, this feature makes programming for local grid refinement much easier since there is no need for a CPU for launching irregularly-sized kernels. Texture memory in Kepler GPUs also has been improved by removing the need to bind and unbind every time a kernel is launched. This is a huge improvement for short-lived kernels such as saturation updates, obtained by reducing the overhead time for binding and unbinding. In addition, in Kepler GPUs, a texture cache pipeline can be used easily via a read-only data cache feature. Since this is a separate cache with a separate memory pipe and with relaxed memory coalescing rules, this feature can benefit the performance of bandwidth-limited kernels (Nvidia 2014).

Section 3. Implementation in CUDA GPU

In this section it will be discussed how to implement IMPES reservoir simulation in the GPU programming language CUDA. The pseudocode of the IMPES simulation is shown in Figure 3.1. In IMPES reservoir simulation, it is not necessary to solve the pressure field for every saturation time step, because pressure changes less rapidly in time than saturation and the stability constraint on the time step size results from the explicit saturation calculation (Chen, Huan and Ma 2006).
The main program (controlling the workflow) is done in the CPU because the GPU is not a general purpose processor and not suitable for this task. However, each of the three parts in this program exhibits a lot of data parallelism that is expected to be a good candidate to be offloaded into and accelerated using a GPU.

In accordance with the reservoir simulation workflow, this section is divided into:

1. Transmissibility matrix assembly, flux interface calculation and saturation update
2. Solving of the linear system of pressure equations

The division of this section into two parts is based on the fact that the transmissibility matrix assembly, flux interface calculation and saturation update are very similar, whereas solving for the pressure involves many different aspects.

### 3.1. Matrix Assembly, Flux Interface Calculation and Saturation Update

In this section, it will be discussed how to implement matrix assembly, flux interface calculation and saturation updates. They are grouped together because all of them have the same type of operation (stencil) and hence allow for similar types of optimization and implementation. The pseudocode for each of the three parts is given in Fig.3.2-Fig.3.3.

#### Figure 3.2 Transmissibility matrix assembly pseudocode.

```
MatrixAssembly:
   For index=1 to noOfGridCells
      Sum=0;
      For index2=1 to noOfAdjacentCell(index)
         TransMat[index,index2]=harAvg(trans(index), trans(index2));
         Sum+= Transmat[index,index2]
      End
      TransMat[index,index]=-sum
   end
```

#### Figure 3.1 Main program pseudo code.

```
Main:
   readSimProperty; // Read Sim property
   transfer initVartoGPU;
   for timestep=1 until nTimestep
      if (timestep==solver timestep)
         AssemblyMatrixKernel; // on GPU
         SolvePressure; // on GPU
      End
   FluxInterfaceCalculation; // on GPU
   SaturationUpdateKernel; // on GPU
   End
   printOutput;
end
```

The main program (controlling the workflow) is done in the CPU because the GPU is not a general purpose processor and not suitable for this task. However, each of the three parts in this program exhibits a lot of data parallelism that is expected to be a good candidate to be offloaded into and accelerated using a GPU.
By looking at the pseudocode, it can be seen that all of them are quite simple. However optimizations are essential to obtain a high performance on CUDA GPUs. The optimizations are:

3.1. Maximizing Exposed Parallelism

In order to attain high performance on GPUs, a certain amount of parallelism is required. There are two reasons why parallelism is required.

1. To provide instructions in flight on both arithmetic pipeline and memory operations

The numbers of cores per streaming multiprocessor (SM) has been increased in Kepler GPUs to 192 compared to 32 in the Fermi GF100. Therefore, according to Little’s law, in order for a Kepler GPU to obtain its maximum ALU bandwidth, it has to be given roughly six times more parallelism than its Fermi counterpart. This parallelism is required in order to provide instructions in-flight in the core pipeline to obtain peak bandwidth.
2. **To dispatch enough instructions for superscalar units per clock cycle.**

   Compared to a Fermi GF100, a Kepler GPU implements in-order dual-issue superscalar to improve its utilization. The superscalar concept implies that Instruction Level Parallelism (ILP) is required for the GPU warp scheduler to fully utilize the GPU by issuing dual instruction per cycle.

   ![](image1.png)

   **Figure 3.6 Superscalar architecture on GF104 vs GF100 (Smith 2010)**

   Parallelism can be expressed in two ways, either via thread level parallelism (TLP) and ILP. TLP is possible because of the GPU’s ability to switch thread with zero overhead, while ILP is possible by dispatching two parallel instructions at one cycle from a single warp scheduler. However, as explained before, because of its superscalar architecture, some degree of ILP is required to achieve peak performance in Kepler GPUs.

   As can be seen from the pseudocode, the calculation of each grid cell is independent of each other, and therefore we can use one thread to obtain the result for one grid cell. This type of organization (one thread for one cell calculation) makes the code more readable and easier to understand than if we use one thread to calculate more than one cell. Also it can be observed that traversing adjacent cells (in the second-stage for-loop) is also an independent operation. Therefore, we can unroll the loop to achieve the required ILP for superscalar architecture.
3.1.2. Coalescing Global Memory Access

Perhaps the single most important performance consideration in programming for CUDA-capable GPU architectures is the coalescing of global memory accesses (Nvidia 2014). The concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of cache lines necessary to service all of the threads of the warp (Nvidia 2014). In Kepler GPUs, L1 is not used to cache global memory load, which is cached only in L2. Therefore the cache line being used is a 32-byte (8 single precision 4-byte float) memory segment. Global memory access can be coalesced by modifying two parts:

a. Changing the data structure from an array of structure (AoS) into a structure of arrays (SoA).

In an AoS, the property of the next grid cell is not located adjacent to the property of the previous grid cell. By changing from an AoS into a SoA, the property of the next grid cell is located in the adjacent memory location. This is visualized in Figure 3.7.

![Array of Structures](image)

**Figure 3.7 Memory location in an AoS versus a SoA (Esler, et al. 2011).**

b. Padding 2-D arrays with extra elements until the width is a multiple of the cache line

By padding the row of 2-D arrays until their width is a multiple of 32-byte, memory access will be coalesced since every row will start with an index that is a multiple of 32-byte. Although the new width (pitch) is larger than the original width which results in more memory storage, the access for the entire row will be coalesced, which will reduce memory access time. The effect of additional padding on 2-D arrays is illustrated in Figure 3.8.

![Structure of Arrays](image)
3.1.3. Reducing Redundant Global memory Access by Using explicit Shared Memory

As can be seen from the pseudocode, to calculate the result for a grid cell, the data from the cell itself and from the four adjacent cells is required. This access pattern is very redundant since the same memory location is accessed five times (by the cell itself and the four neighboring cells).

In Fermi GPUs, global memory load and write is automatically cached in L1, and therefore, usually there is no need to explicitly use shared memory (Maruyama and Aoki 2014). However, in Kepler GPUs, the only high-bandwidth, low-latency, on-chip memory that is able to store temporary data is shared memory. Hence, in order to avoid redundant long latency access to global memory, each streaming multiprocessor (SMX) shared memory must be used explicitly. Since shared memory is only accessible to a thread member within a thread block, it is advised to use as large as possible thread blocks to increase memory sharing opportunities and reduce redundant global memory access. In Kepler GPUs, the maximum thread block size is 32x32 threads. The division of our problem domain into thread blocks is shown in Figure 3.9.

![Figure 3.9 Division of simulation cells into thread blocks (Cooper 2011)](image)

However, care should be taken in prefetching the data into shared memory. Loading ghost points (at boundaries) to the shared memory should be done with warp specialization. Warp specialization means that every thread in a warp (a set of 32 threads) is doing the same things without divergence in execution. This is essential because if the threads in a warp are coherent, the performance will be high because they are executed simultaneously. The pattern for loading ghost cells is displayed in Figure 3.10.
By using warp specialization, the divergence execution can be avoided, but a memory bank conflict incurs because every thread in a warp is accessing the same column element, which has the same shared memory bank and which will therefore serialize the memory access. The solution is simple: just pad the two-dimensional shared memory with an extra column as illustrated in Figure 3.11.

Because shared memory is used as a fast memory to cache data from global memory, the program workflow should be modified into:

1. Load Grid Cell values of a block (from low-bandwidth global memory) into high-bandwidth shared memory
   The data is being loaded to shared memory because a grid cell values will be used five times, hence by using fast shared memory, redundant access to slow global memory is avoided.
2. Wait until the whole threads in a block has fetched the data.
3. Calculate Flux interface using high bandwidth shared memory
   The calculation is being performed using data that come from fast, on-chip shared memory, hence the data bandwidth is able to keep up with the core bandwidth. Furthermore, the flux calculation is also stored in shared memory for further explicit saturation update calculation.
4. Wait until all threads in a block finish calculate the flux
5. Update the saturation from data using flux from shared memory
It should be noted that the prefetching technique is powerful in a superscalar architecture (e.g. Kepler). This is because non-superscalar architectures, (which are only able to dispatch an instruction per thread per cycle), cannot issue load and calculation instructions from a thread together in a cycle, thus making load units idle while calculating and vice versa. Instead, in superscalar architectures, the
loading and calculation instructions can be issued together, hence while some of the threads are prefetching data from global memory to shared memory, some threads can do calculation of other blocks that have already been in shared memory.

### 3.1.4. Using pre-computed table compute intensive function

Relative permeability function evaluations are very expensive to compute because they usually involve a combination of functions (such as trigonometric functions and polynomials) which require a lot of cycles (i.e. a low throughput) to compute. Therefore a good practice is to create pre-computed lookup tables of relative permeabilities for a certain saturation range. Hence, there is no need to calculate relative permeability functions every time step for every grid cell, and the code just has to look for the relative permeability value in a pre-computed lookup table. This technique is very powerful especially in GPU programming because GPUs are able to hide memory access latency by switching to other calculation-ready threads without overhead (zero overhead thread switching).

Since the saturation range is not large, a lookup table is pre-computed in the CPU and then transferred to GPU at the beginning of execution. CUDA memory that provides a good fit for the pre-computed lookup table is 1-D texture memory. This is because it is cached on-chip, good for scattered or randomized access and can interpolate linearly between values with little additional computation time. Furthermore, in Kepler GPUs there is no need to bind and unbind texture memory every time a kernel is launched, and therefore the overhead for using texture memory is eliminated. For IMPES simulations, this feature is beneficial since the kernel in IMPES is short-running which makes even a little overhead undesirable.

### 3.1.5. Using Various CUDA Read-Only Memories

In IMPES simulations, there are several types of read-only data such as simulation parameters (e.g. reservoir size, time step size, etc.), saturation values at the previous time step, and grid cell permeabilities. Since they are read-only data, global memory bandwidth usage can be reduced by using the special cache provided by GPUs to access special types of read-only data. However, the access patterns of these data are not all the same. Therefore, they should be put in and accessed by different hardware caches in the GPU. The access pattern and optimal cache location is described in Table 1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Access Pattern</th>
<th>Optimal Location</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation parameters</td>
<td>Each threads access the same data</td>
<td>Constant memory</td>
<td></td>
</tr>
<tr>
<td>Grid Cell Permeability,</td>
<td>Each threads access data with 2D Spatial locality</td>
<td>Read-only data cache</td>
<td></td>
</tr>
<tr>
<td>Saturation at previous timestep</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.2. **Solver for Linear System of Pressure Equations**

In this section it will be discussed on how to solve the linear system of pressure equations, particularly the equations arising in foam EOR simulations, using CUDA.

3.2.1. **Solver and Preconditioner**

The transmissibility matrix that arises from the pressure equation has several properties such as:

1. Sparseness
2. Ill conditionedness (Chen, Huan, & Ma, 2006)
3. Huge dimension (1e6 grid cells will create a matrix with a total size of 1e12)
4. Naturally bandedness (for structured grid)
5. Symmetry and positive definiteness

Therefore, for such a matrix system, Krylov subspace iterative algorithms are the ideal option (Chen, Huan, & Ma, 2006). Furthermore, a preconditioner is also implemented because Krylov subspace methods are often useless without preconditioning (Chen, Huan, & Ma, 2006).

There are several variations of the Krylov subspace algorithm such as conjugate gradient (CG), BiCGSTAB (a biorthogonalization method adapted from the CG method), etc. In choosing an iterative solver, there is no different guideline for CPUs and GPUs because the solver itself relies on sparse matrix vector multiplication (SPMV) which also works well on GPUs. Hence, the chosen solver is based only on the matrix type. Because the matrix is symmetric and positive definite, the chosen solution method is CG.

The chosen preconditioner should not only be suitable for parallel processing, such as in GPUs, but also be able to reduce the solver time to its minimum. Because of this, using a complex preconditioner will not necessary result in minimum time because preconditioning itself could also take a long time. Based on our testing, using a Jacobi preconditioner, the iteration number per time step is roughly less than 15 for a million grid cells. Therefore, it is concluded that a Jacobi preconditioner is good enough to reduce the iteration count, and there is no need for a more complex preconditioner. Furthermore, a Jacobi preconditioner is excellent for GPUs because of its highly parallel nature.

3.2.2. **Sparse Matrix Format**

The matrix that arises from the spatial discretization of the pressure equation has a large dimension, usually more than thousands by thousands in size. However, it is also sparse, since in the flow equation one cell interacts only with its direct neighbours. A good sparse matrix format doesn't only reduce the space requirement, but also enables faster sparse matrix operations. Moreover, since most of the reservoir simulation time is spent on solving sparse linear system equations (Appleyard, Appleyard, Wakefield, & Desitter, 2011), choosing a good sparse matrix format is essential in reservoir simulation to facilitate faster sparse matrix solvers. In GPUs, this is achieved by the format that gives a high memory throughput via coalesced access memory.
There are several sparse matrix formats such as the coordinate (COO), compressed sparse row (CSR), diagonal, Ellpack (ELL) or hybrid formats. There is no single 'best' sparse matrix format for all cases (Kumbhar, 2011). However we can choose the format based on the sparsity pattern of the matrix that we are dealing with.

The matrix that arises from a two-dimensional structured grid is a penta-diagonal matrix. For a perfectly pentadiagonal sparse matrix, the best sparse matrix format is a diagonal format since it gives higher memory throughput, flops and hence lower times in SPMV and iterative linear solver operations. (Kumbhar, 2011).

![Sparsity Structure](image)

**Figure 3.12 Performance of various sparse matrix formats for banded matrices.**

However, the diagonal format doesn't perform well when there is a simple deviation of a perfectly diagonal format. Therefore, in case of local grid refinement, which will create deviations from perfect pentadiagonal format, an ELL or hybrid format will give better performance (Kumbhar 2011).

The diagonal format uses two arrays to store its value. The first array indicates which diagonals are filled, and the second one stores the non-zero values. The array-value is stored using a column major format with offsets and diagonal index in 32-bit multiples (L1 cache line size). This storage format allows a coalesced access pattern which will give faster memory throughput in GPUs.

![Diagonal sparse matrix format](image)

**Figure 3.13 Diagonal sparse matrix format.**

**Section 4. Result and Analysis**

In this chapter we will discuss the result of using a GPU in comparison to a CPU for a numerical example of a 2-D foam EOR simulation. The details of the reservoir model are provided in Appendix A. We considered four different reservoir models.
with $10^5$, $10^6$, $6\times10^6$ and $7\times10^6$ gridcells respectively. The hardware used for the comparison consisted of:

1. An Intel core i7 4770K (Intel 4th generation core i7 flagship CPU)
2. A Nvidia GeForce Gt650m GPU in a MacBook Pro retina (a Kepler GPU with 384 cuda cores)
3. A Nvidia Geforce GTX Titan Black SC (a Kepler GPU with 2880 cuda cores)

The CPU implementation was implemented in the C programming language, and parallelized with OpenMP. To obtain a fast CPU performance, the code was compiled using –O2 flag in Visual C++ to optimize the code.

The CPU foam simulation timing for a model with $10^5$ grid cells is depicted in Figure 4.1. As can be expected, most of the reservoir simulation time (73%) is spent on the pressure solver. However, the time spent on the saturation update and matrix assembly (27%) cannot be neglected either, especially if the pressure is not solved for every saturation time step. Hence, it would be beneficial if both parts could be accelerated using the GPU.

The resulting speedup of using GPU is tabulated in Table 2.

**Table 2. Resulting GPU speed-up**

<table>
<thead>
<tr>
<th>Grid Cells</th>
<th>GT650m Speedup [x]</th>
<th>Titan Speedup [x]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>over CPU</td>
<td>over CPU</td>
</tr>
<tr>
<td>$1.00E+05$</td>
<td>10.18</td>
<td>87.46</td>
</tr>
<tr>
<td>$1.00E+06$</td>
<td>12.18</td>
<td>122.60</td>
</tr>
<tr>
<td>$6.00E+06$</td>
<td>10.82</td>
<td>127.82</td>
</tr>
<tr>
<td>$1.00E+07$</td>
<td>11.00</td>
<td>129.42</td>
</tr>
</tbody>
</table>

**Pressure Solver**

<table>
<thead>
<tr>
<th>Grid Cells</th>
<th>GT650m Speedup [x]</th>
<th>Titan Speedup [x]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>over CPU</td>
<td>over CPU</td>
</tr>
</tbody>
</table>

![Time Percentage 1e5 Grid Cells](image)

**Figure 4.1 Time percentage in a 1e5 grid-cell CPU foam EOR simulation.**
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00E+05</td>
<td>3.371</td>
<td>4.100</td>
<td>1.216</td>
</tr>
<tr>
<td>1.00E+06</td>
<td>6.995</td>
<td>22.338</td>
<td>3.193</td>
</tr>
<tr>
<td>6.00E+06</td>
<td>7.473</td>
<td>38.466</td>
<td>5.147</td>
</tr>
<tr>
<td>1.00E+07</td>
<td>7.290</td>
<td>39.049</td>
<td>5.356</td>
</tr>
</tbody>
</table>

**Analysis**

It can be seen that the characteristic results of both parts are quite different. Therefore both parts are discussed separately.

1. **Pressure Solver**

There are several important aspects that can be inferred from the result.

1. **Titan black vs GT650m GPU speedup is \( \approx 5.35 \) at max**

For large models (>6 millions grid cells), the speedup is dominated by the global memory bandwidth difference between Titan black and Gt650m GPUs. The result is in accordance with our expectations, because iterative solver is expected to be global memory bandwidth-limited. This is because the iterative solver relies on SPMV and vector reduction, two operations that are well known to be memory bandwidth-limited (Bell and Garland 2008), (Harris 2007). Therefore, it can be expected that the speedup of the pressure iterative solver is in accordance with the GPU memory bandwidth.

2. **The Titan black vs GT650m GPU vs CPU speedup is small for small models**

For small models the speedups between the Titan black GPU, the GT650m GPU and the CPU are quite small. This behavior can be attributed to two aspects:

1. **For smaller models, the data fits completely in the CPU cache**

   The amount of L3 cache in the Intel 4770k CPU is 8Mb. This means that the cache could fit up to 2 million single precision floating point numbers. Therefore, all of the data in a 1e5 grid cell model fits on the CPU cache. This makes the program fast because the CPU can cache all the data and make the subsequent data access much faster than if it would fetch from RAM.

2. **For smaller models, latency between the CPU and GPU dominates runtimes**

   It can be seen that for smaller models, the Titan Black is not much faster than the Gt650m. Furthermore, the difference in speed between the Titan black and the Gt650m grows with larger models and peaks at \( \approx 6e6 \) grid cells.

   This phenomenon might be attributed to the latency between CPU and GPU. This is because the workflow of the solver itself is done in the CPU while using the GPU as a co-prosessor to do the parallel tasks (SPMV and reduction). Hence, there is still a data transfer to the CPU for decision making. Although the amount of data transferred is small, the latency
between the CPU and the GPU could be significant compared to the calculation runtime for small models.

2. Saturation Update and Matrix Assembly

The speed differences between the Titan Black GPU, the Gt 650m GPU and the CPU are very high (speedup factors of 11.76 and 129 respectively). These results show that the program is not global memory bandwidth-limited as in the pressure solver. This performance can be attributed mainly to the optimization using various GPU memories as explained in section 1. The optimization is possible since there are a lot of data reuse opportunities in the program which can be cached in high bandwidth, low latency, on-chip memory (register, shared and texture memory) for fast subsequent access. Since the on-chip memory is expected to be able to keep up with the GPU computational speed, using them will result in a high performance program.

Section 5. Conclusions

There are several conclusions that can be drawn from this study:

1. Compared to the speedup obtained with Fermi GPUs (Esler, et al. 2011), Kepler GPUs improve the computational performance of IMPES reservoir simulation even further. In our implementation, the improvement originates not only from the saturation update step but also from the pressure solution step.

2. Additional optimization is required when using Kepler GPUs to achieve a high performance. Especially because automatic L1 caching has been removed in Kepler GPUs, explicit shared memory optimization needs to be implemented in order to exploit high-bandwidth and low-latency on-chip memory. Furthermore, an efficient saturation update kernel needs to be implemented to express instruction level parallelism as a requirement to achieve peak performance in Kepler GPUs. Finally, various cuda read-only memories can be used to reduce global memory traffic and improve the speedup even further.

3. Using GPUs, and an appropriate implementation of the simulation code, the run time of large-scale foam simulations (using models with 6 million grid cells or more) can be reduced from months to days.

Section 6. References


Appendix

A. Foam Simulation Results

In this section we will show the result of foam simulation. The parameter of the simulation are shown in Table A-1

<table>
<thead>
<tr>
<th>Property</th>
<th>Values</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reservoir Length</td>
<td>2500[m]</td>
<td></td>
</tr>
<tr>
<td>Reservoir Width</td>
<td>2500[m]</td>
<td></td>
</tr>
<tr>
<td>Grid Cell sizes</td>
<td>1x1 [m.m]</td>
<td></td>
</tr>
<tr>
<td>Injection Well</td>
<td>2 at (0m,0m) and</td>
<td>Constant injection rate policy, 100bpd each.</td>
</tr>
<tr>
<td></td>
<td>(0m,2500m)</td>
<td>Injector composition (Sg=1)</td>
</tr>
<tr>
<td>Producer Well</td>
<td>1 at (2500m,2500m)</td>
<td>Constant bottom hole pressure, 1e7 Pa. The rate is modeled by peaceman</td>
</tr>
<tr>
<td></td>
<td></td>
<td>well model for rectangular grid.</td>
</tr>
</tbody>
</table>
|                        |                   | \[
|                        |                   | \[q = \frac{2\pi h k_r \tilde{p} - p_w}{{\mu} \ln(\frac{r_e}{r_w} - \frac{1}{2})}\] |
|                        |                   | with,                                                                |
|                        |                   | \[h =\text{reservoir height}\]                                      |
|                        |                   | \[k_r = \text{fluid relative permeability}\]                         |
|                        |                   | \[k = \text{cell permeability}\]                                    |
|                        |                   | \[\tilde{p} = \text{cell pressure}\]                               |
|                        |                   | \[p_w = \text{well bottom hole pressure}\]                          |
|                        |                   | \[r_e = \text{grid cell width}\]                                    |
|                        |                   | \[r_w = \text{well radius}\]                                        |
| Reservoir Model        | 6.25x10^6 cells, permeability given in Figure A.1                  |

In the absence of foam we used the following relative-permeability functions for water and gas:

\[
k_{rw}^0 = 0.6822 \left( \frac{S_w^{0.05}}{0.90} \right)^{2.6844}
\]

……………………………………………………………………………[A.1]

\[
k_{rg}^0 = 0.8649 \left( \frac{0.95 - S_w}{0.90} \right)^{2.2868}
\]

……………………………………………………………………………[A.2]

Water and gas viscosities are 1.0 and 0.02 cp respectively. With foam, water relative permeability function and viscosity are not altered but gas mobility is greatly reduced (Rossen 1996). We use foam model in STARS™ (Cheng, et al. 2000), in which the effect of foam is represented by an alteration in gas relative permeability.
$$k_{rg} f = \frac{k_{rg}^0 (S_w)}{1 + f_{mmb} (0.5 + \arctan (e_p_dry \cdot (S_w - f_{mdry})) \pi)}.$$

with parameter for the foam alteration is $f_{mmb}=34,000$, $f_{mdry}=0.13$ and $e_p_dry=10,000$.

In order for the relative permeability to be cached, it is pre-computed for 1000 saturation values. In the program, if the requested values is not on the table, the result will be interpolated linearly between two closest saturation values.

The simulation result is given in Figure A.2 for saturation profile and Figure A.3 for pressure profile.

Figure A.1 Reservoir permeability map

Figure A.2 Saturation profile in the reservoir

Figure A.3 Pressure profile in the reservoir
**B. Nvidia CUDA GPU**

In order to make an efficient and robust reservoir simulator using CUDA Graphics Processing Units (GPU) accelerator, knowledge of CUDA GPU is very important since it is architecturally and programmed very different from general purpose CPU. Therefore, in this section, essential knowledge of CUDA GPU, from the architecture, programming models and also optimizations is discussed. Note, since the GPU architecture evolves very rapidly, this discussion is limited to current generation (*Kepler*) GPU, unless stated otherwise.

CUDA GPU accelerator is a device, used to accelerate application that exhibits high arithmetic operation density (ratio of arithmetic to memory operations) and data level parallelism. Data parallelism means that a single instruction is executed for a lot of different data which can be executed in parallel. GPU is a separated device from CPU which has its own memory and processor. In INTEL sandybridge platform, GPU is connected via PCI-E 2.0 bus which has bandwidth 5Gb/s each way as illustrated in Figure B. The common programming practice is to transfer part of the program which is suitable and meant to be accelerated in GPU and then transfer the result back to CPU to be analyzed further.
B.1. CUDA GPU architecture

The architecture of GPU accelerator is very different from general purpose CPU that is usually used. GPU architecture is driven by insatiable market demand for realtime, high-definition 3D graphics which makes it highly parallel, multithreaded, many core that offers tremendous computational horsepower and very high memory bandwidth (Nvidia 2013). The tremendous high floating-point operations per second (FLOPS) and memory bandwidth of GPU compared to CPU is illustrated in the Figure.
GPU accelerator has a very high FLOPS because it dedicates much of its transistor to data arithmetic logical unit (ALU) compared to CPU which gives a lot of transistor to cache and flow control as illustrated in Figure. 

The architecture of the current generation GPU (Kepler) is described in Figure B.
A single GPU consist of several parts such as:

a. **Single Global Device Memory**
   
   Every GPU has a global device memory. The global memory has a size usually around 3 Gb depends on GPU. Global memory has a bandwidth $\approx 200\text{Gb/s}$ and latency $\approx 400$ to $800$. Although it is faster compared to CPU’s random access memory (RAM), it is very slow compared to GPU FLOPS. However, it is the largest memory available in GPU, accessible to CPU and all of the threads in GPU at anytime.

b. **Several Streaming Multiprocessor**
   
   In every GPU there are several streaming multiprocessors (SMX) ranging from 2 in GeForce Gt650M until 15 in GeForce GTX 780 Ti, depending on the type. SMX is important since every thread in a block (threads collection) is executed on a single SMX. The structure of SMX is described in picture below.

![Figure B.5 Architecture of Kepler SMX (Nvidia 2012)](image)

A single SMX consist of:

a. **192 Single Processors (SP).** In GPU, arithmetic operation is conducted in an SP. It is similar to single core in multicore CPU.

b. **65536 registers.** In GPU, register is used as a very fast ($\approx (8Tb/s)$ with 1 clock latency) memory which is private for every thread.

c. **64 Kb Shared Memory/ L1 Cache.** In Cuda, Shared memory is fast memory ($2Tb/s$, 1 clock latency) which can be used to share data between threads in a thread block. This memory is very useful in reservoir simulation to reduce global memory access, as it is shown later. Shared memory can be accessed using `__shared__` keyword. The L1 cache, is used as backup memory when more register is needed.

d. **Read Only Data Cache**
This cache is a new addition to the last generation GPU. It is used to store read-only data which can be accessed randomly between threads with an access cost of coalesced global memory access. Hence, it is useful to store data in this cache such as lookup table when it is accessed randomly for every thread. The size of this cache is quite big, 48KB per SMX. This cache can be accessed by using \textit{const} and \texttt{__restrict__} keyword.

\subsection{Constant Memory}
In every device there is a total of 64 KB constant memory cache (Sanders & Kandrot, 2011). The constant memory is aimed for read-only data which will be broadcasted to the whole thread. So, this memory is excellent when every thread access the \textit{same} location of memory at the same time. However, if different threads in a \textit{warp}, fetch \textit{different} data in constant memory at the same time, the memory access will be serialized, and the performance will suffer.

\subsection{Texture Memory}
Texture memory is a read-only memory which basic designed is for graphic processing. In general computation, it is beneficial to reduce global memory bandwidth when data access involves spatial locality between threads such as described in the picture below.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{texture_memory.png}
\caption{Data access that exhibits spatial locality (Sanders & Kandrot, 2011)}
\end{figure}

\subsection{L2 Cache}
L2 cache is 1536Kb memory that has a faster speed than global memory. It is available to every thread in GPU, so it could be used as fast way to interact between threads in different block or SMX.

\section{B.2. CUDA GPU programming models}
In November 2006, NVIDIA, introduces \textit{CUDA\textsuperscript{TM}}, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU (Nvidia 2013). CUDA is aimed to simplify general purpose programming code for GPU in comparison with common graphics programming language such as Microsoft DirectX, etc.

CUDA programming languages is designed for a heterogeneous computing model, where there is a \textit{host} which is usually CPU and \textit{device} which is GPU accelerator. This means that it can separate code which is meant for CPU (serial and complex) and GPU (data level parallel). The CPU code is just a regular C or C++ code which will be compiled via regular C/C++ compiler such as GNU compiler \textit{GCC} or Intel compiler \textit{ICC}, while the code for GPU is C/C++ semantic code with additional added keywords specific for GPU architecture. The heterogeneous programming model can be described with the picture below.
1. **Thread and memory Hierarchy**

In CUDA programming model, thread hierarchy consists of thread, block and grid. Thread is the smallest form of independent process in CUDA. Each thread is executed on a Single Processor. Furthermore, every thread also has its own register for storage. Thread has 3-component vector index for identification. Therefore, threads can be identified by one, two or three-dimensional index. Since every thread has its own register, CUDA has a powerful feature that CPU doesn’t have. The special feature is zero overhead thread switching, which means that switching from a thread to another cost nothing. This feature is very powerful when a thread is waiting for memory access, hence the latency for memory access can be hide by computation of other thread.

Block is a collection of threads. The maximum number of threads per-block is limited by GPU generation (current Kepler GPU can accommodate up to 1024 threads per block). Thread block is a special grouping since all of the threads in a block is executed in a single SMX. Furthermore, since every thread in a block is executed on the same SMX, they can communicate with each other. They can communicate (send/use data) via fast shared memory which is available for every SMX. CUDA also provides means of barrier of threads in a block in order to avoid *race condition* between threads. This can be accessed via __syncthreads() procedure.
On the other hand, every block is completely independent with each other because there is no guarantee that each of them will be executed in the same SMX and at the same time. Therefore, the independency between each thread blocks provides excellent means of scalability between different types of GPU. The thread and memory hierarchy of CUDA programming can be described via picture below.

2. **CUDA Program Execution Model**

CUDA program is executed in a single instruction multiple thread (SIMT) fashion. This means that every cores (SP) is executing the same instruction at the same time. To be more precise, 32 threads are grouped into a warp. In GPU, a warp is executed in a lockstep fashion. This means that every thread in a warp executes same instruction at the same time. This model is described in the picture below.
This programming model has an advantage that it requires less instruction controller which provides more chip area to arithmetic logical unit (ALU); however, if there is a branch execution between thread in a warp, it will be executed sequentially since every thread in warp is executed in a lockstep basis. Therefore, any divergence in a warp will create an idle SP and reduce performance since thread in a warp cannot execute different instructions. Thus, any divergence between thread in a warp should be avoided.

B.3. CUDA GPU program Optimization

By using CUDA programming language it is easy to make program that works on GPU, however, to fully utilize the GPU optimally, there are a lot of things to optimize in CUDA program. The optimizations are:

1. Memory Optimization

Memory optimization is very vital in CUDA programming, especially in the program like reservoir simulator where the ratio of computation to memory access is not high. Moreover, from various optimization methods, memory optimization usually gives the most improved result and it is also the easiest to implement.

Let’s take a look why memory optimization is important in CUDA. Nvidia G80 supports 86.4 Gb/s global memory access bandwidth. With 4 bytes in each single floating-point load, one can expect no more than 21.6 billion FLOPS calculation if every calculation needs one memory load. Although it seems a huge number, it is very far from maximum FLOPS of Nvidia G80 can do which goes up to 367 gigaFLOPS (Kirk & Hwu, 2010).

Various methods to optimize memory access in CUDA are:

a. Maximize the occupancy (active threads divided by maximum threads per SMX) of an SMX

Recall that GPU can switch threads at no cost, when the thread is stalls (wait for memory access). Therefore switching from a stall warp into a ready warp is a good way to hide memory latency access as described in Figure B. below. Current generation CUDA (Kepler) can track and manage up to 2048 threads per SMX. Therefore if we can give full occupancy to every SMX there are a lot of threads to hide the latency of other threads in an SMX. To maximize occupancy it is the job of programmer to choose block size and reduce SMX usage resources such as register usage which will maximize the occupancy of SMX. The guideline for block size number for each generation CUDA is given in the Figure below.
b. Coalesced memory access

In Kepler GPU, the minimum size of memory access is 128 bytes which corresponds to 32, 4-byte floating point. Therefore if all of the threads in a warp (32 threads) access memory in that 128 bytes segment, it will only lead to only one memory fetch latency cost. In implementation, it is a common practice for 2D array to be padded into nearest 32 multiples, so it can be accessed coalescedly. This optimization can be seen in our simulator later on. The bandwidth improvement of coalesced memory access on older GPU can be seen in figure below
c. Using Shared memory for data reuse and sharing intra-block
   Recall that shared memory is hundred times faster than global memory, therefore if a thread want to share data that it has been computed to other threads, shared memory is a good way to minimize global memory access.

d. Use other parts of CUDA memory for special features.
   There are a lot of cache in CUDA GPU which provides special features like constant data cache, constant memory and texture memory. All of them have special functionality that they are good at. Constant memory is good when all of the threads access the same memory location. While constant data cache is good for scattered randomized read only data such as lookup table. Last, texture memory is good when the data access exhibit close spatial memory location.

e. Reduce CPU to GPU memory access or access it asynchronously.
   Transfer data from CPU to GPU is very slow (5GB/s one way). Therefore a good practice is to reduce data transfer as much as possible. Another way is to do it asynchronously, transferring the data while CPU and GPU doing useful work.

2. Branching optimization
   Besides memory optimization, there are several possibilities to optimize GPU program performance. Minimizing branching intra warps is one of the most essential optimization in CUDA. Recall that CUDA program is executed in a lockstep basis for every warp (32 threads), therefore if there is any divergent (different path) of threads within warps, the program will do all of the path sequentially. This will leads to inactive SP and therefore reduce performance. Nevertheless it is not a problem to make a divergent path between warp, since every warps is executed differently. This can be illustrated with a code fragment below which describes problematic branching and not.
C. Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Block</td>
<td>A group of threads that is executed on the same SM/SMX and can share the same SM/SMX resources</td>
</tr>
<tr>
<td>Fermi</td>
<td>2nd generation nvidia GPU architecture</td>
</tr>
<tr>
<td>FLOPS</td>
<td>Floating point operations per second</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction level parallelism. Parallelism in the form of many independent instructions per thread</td>
</tr>
<tr>
<td>IMPES</td>
<td>Reservoir simulation method where pressure is solved implicitly and saturation is updated explicitly</td>
</tr>
<tr>
<td>Kepler</td>
<td>Current (3rd) generation nvidia GPU architecture</td>
</tr>
<tr>
<td>L1 cache</td>
<td>High bandwidth, low latency on chip memory that is automatically managed by hardware as a copy of off chip, low bandwidth global memory</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>The rate of memory in delivering data (Gb/s)</td>
</tr>
<tr>
<td>Memory latency</td>
<td>Lag time between when data is requested and the time it is arrived</td>
</tr>
<tr>
<td>SM/SMX</td>
<td>Streaming multiprocessor. A set of cores that are being grouped together. Every cores in SM can share the usage of its shared, on chip resources such as fast shared memory, texture memory etc.</td>
</tr>
<tr>
<td>Superscalar</td>
<td>Computer architecture that enables the execution of more than one instruction per cycle by dispatching more than one independent instructions per thread in a clock cycle.</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread level parallelism. Parallelism in the form of many independent threads</td>
</tr>
<tr>
<td>thread 1</td>
<td>thread 2</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>( x = x + c )</td>
<td>( y = y + c )</td>
</tr>
<tr>
<td>( x = x + b )</td>
<td>( y = y + b )</td>
</tr>
<tr>
<td>( x = x + a )</td>
<td>( y = y + a )</td>
</tr>
</tbody>
</table>

**Warp**

A group of threads that is executed in a lockstep basis. Different execution paths should be avoided within the same warp because it will be executed serially.