Mejoramiento del tiempo de ejecución de un núcleo Lattice-Boltzmann utilizando la arquitectura NVIDIA G80

Improving the execution time of a Lattice-Boltzmann kernel using the NVIDIA G80 architecture

SNAIDER CARRILLO
Electronic Engineer from the Universidad del Norte (Barranquilla - Colombia). Master of Science in Electronic Engineering from the Pontificia Universidad Javeriana (Bogotá - Colombia). Associate Professor in the Department of Electronic Engineering and Telecommunications at the Universidad Autónoma del Caribe (Barranquilla - Colombia). Correo electrónico: snaidercl@ieee.org

JAKOB SIEGEL
Software Engineer from the HS Essligen: University of Applied Science (Essligen - Germany). PhD Candidacy in Electrical and Computer Engineering at the University of Delaware (Newark - USA). Correo electrónico: jsiegel@udel.edu

XIAOMING LI
B.S. and M.E. in Computer Science from Nanjing University (Shanghai - China), and Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign (Urbana - USA). Assistant Professor in the Department of Electrical and Computer Engineering at the University of Delaware (Newark - USA). Correo electrónico: xili@udel.edu

Clasificación del artículo: Investigación (conciencias)

Fecha de recepción: 29 de septiembre de 2009 **Fecha de aceptación: 21 de enero de 2010**

Palabras claves: Arquitectura NVIDIA G80¸ branch-splitting, T, Optimización a nivel de instrucciones.

Keywords: Branch splitting, instruction level optimization, *Loop splitting*, NVIDIA G80 architecture

RESUMEN

Las sentencias de control utilizadas en el entorno de programación de las tarjetas graficas de video (GPUs), tales como sentencias condicionales e iterativas, presentan problemas de concurrencia al momentos de ser ejecutadas debido a que tienden a minimizar el nivel de ocupación, el cual es una medida que nos permite establecer el número de procesos que se están ejecutando de forma concu-

rrente en las GPUs. A diferencia de las unidades de procesamiento de datos tradicionales presentes en un microprocesador de propósito general, una GPU no puede ceder el control de flujo de datos a la CPU debido a que actualmente no existe un mecanismo que permita dicho control de flujo sin comprometer la integridad de los datos procesados en dichas arquitecturas.

En este artículo se proponen y evalúan dos nuevas técnicas de optimización a nivel de instrucciones enfocadas a hacer un mejor uso de los recursos de tipo hardware de las GPUs, en la arquitectura NVDIA G80. Estás técnicas llamadas *Loop splitting* and *branch splitting* incrementan de forma controlada la redundancia de código, lo cual puede ser considerado como “no optimo” en una arquitectura convencional como la CPU; sin embargo en la arquitectura multiprocesador NVIDIA G80 dicha redundancia se ve reflejada en el incremento de la ocupación de sus multiprocesadores y en un aumento del paralelismo de los programas ejecutados en este tipo de arquitectura. Los resultados obtenidos a partir del banco de pruebas, basados en el algoritmo del método de Lattice Boltzmann (LBM), muestra que estas técnicas incrementan la ocupación y el paralelismo de la arquitectura NVIDIA G80 comparado con la ejecución de la versión *non-splitting* del mismo algoritmo.

Abstract

Control statements in a Graphic Processing Units (GPU) program such as *loop*s and *branches* pose serious challenges for the efficient usage of GPU resources because those control statements will lead to the serialization of *thread*s and consequently ruin the occupancy of GPU, that is, the number of *thread*s running concurrently. Unlike traditional vector processing units that are inside a general purpose processor, the GPU cannot leave the control statements to the CPU because fine-grain statement scheduling between GPU and CPU is impossible.

In this paper, we propose novel techniques to transform control statements so that they can be executed efficiently on a GPU called NVIDIA G80. Our techniques smartly increase code redundancy, which might be deemed as “de-optimization” for CPU, to improve the occupancy of a program on GPU and therefore improve performance. We focus our attention on how common programming structures such as *loop*s and *branches* decrease the occupancy of single kernels and how to counter that. We demonstrate our optimizations on a benchmark for a complex parallel algorithm, the Lattice Boltzmann Method (LBM). Our results show that these techniques are very efficient and can lead to an increase in occupancy and a drastic improvement in performance compared to non-split version of the programs.

* * *

1. Introduction

GPUs have become the most powerful computation devices in modern of-the-shelf PCs. Until recently, it was a challenge to implement an algorithm efficiently to run on a GPU because the functionality of such a device was plainly geared toward graphics acceleration, and didn’t offer an interface to perform non-graphics operations. The introduction of the Compute Unified Device Architecture (CUDA) programming framework [1] makes the computational power of GPUs easier to utilize. However, even though the problem of writing a program that can *work* on a GPU seems to have been solved, the question of how to tune a program to make it *work well* on a GPU is only rudimentary understood and insufficiently investigated.

A detailed description about CUDA and its su-

pporting hardware can be found in [1]. Here we briefly discuss the most important factors that impact the performance of CUDA programs. The factors are new and unique to CUDA, hence are not considered in the traditional CPU optimization techniques.

The CUDA defines a new architecture called SIMT (single instruction, multiple-*thread*) [1] which allows a multiprocessor GPU chip map an individual *thread* to one scalar processor core, and each scalar *thread* executes independently with its own instruction address and register state. This is a cost-effective hardware model to exploit data parallelism. The SIMT architecture can be ineffective for algorithms that require diverging control flow decisions, such as those generated from *if-else* statements, because the concurrency among *thread*s will be reduced if *thread*s within the same *thread* group (called *warp* in CUDA) follow different *branches* [2].

One of the main tasks of optimizing a program for CUDA is find the optimal numbers of *thread*s and blocks that will keep the GPU fully occupied. Factors affecting the resource occupancy include the size of the global data set, the maximum amount of local data that blocks of *thread*s can share, the number of *thread* processors in the GPU, and the sizes of the on-chip local memories [1]. One important limit of occupancy of a program is the number of registers each *thread* of the program requires. For example, to reach the maximum possible number of 12,288 active *thread*s in a 128-processor GeForce 8, the compiler cannot assign more than 10 registers per *thread*. However, the CUDA compiler usually over assigns registers per *thread*, which decrease the occupancy of the kernel, because the CUDA compiler tries to optimize the single-*thread* performance while ignoring the overall resource pressure of a multi-*thread* program.

This paper presents new instruction level transformation techniques that improve the utilization of hardware resources of the NVIDIA CUDA platform. Our techniques are novel applications of seemingly common program transformations. In other words, they smartly increase code redundancy, which might be deemed as "de-optimization" for CPU, to improve the occupancy of a program on GPU and therefore improve performance.

The rest of the paper is organized as followed. In Section 2 we present the proposed approach to improve the occupancy and parallelism for the NVIDIA G80 architecture. In sections 3 we evaluate the performance of our optimizations on a benchmark based on the Lattice Boltzmann Method. The conclusions are given in Section 4.

2. Proposed *loop* and *branch* optimization techniques

Optimizing a CUDA kernel for better occupancy allows for better usage of the devices computational resources and better hiding of memory latency, and usually gives a better performance. The basic idea of our techniques is freeing hardware resources by purposefully *increasing code* size by *splitting* common control structures.

2.1. Loop splitting

Loop splitting or *Loop fission* is a simple optimization that breaks a *loop* into two or more smaller *loops*. *Loop splitting* is especially useful for reducing the register pressure of a CUDA kernel, which can be translated to better occupancy and overall performance improvement. If a kernel contains a *loop* where in the *loop* body multiple operations are performed and each operation relies on different inputs and those operations are independent, this optimization can be applied. The *splitting* leads to smaller *loop* bodies and hence reduces the *loop* register pressure. Therefore, this optimization is applicable to kernels that don't reach 100% occupancy because of register usage. *Figure 1 left*, shows a pseudo code segment from a CUDA kernel where we can split the *loop*. After *splitting*, only *ptr1* and *ptr2* have to be kept in registers for the first *loop* and *ptr3* and *ptr4* for the second *loop*. This can be done because all the pointers are parameters passed to the kernel and if we only use those parameters in the *loop* body, we don't have to load the data into registers before the actual usage. Therefore *ptr1* and *ptr2* are getting loaded into registers when the first *loop* is

executed and *ptr3* and *ptr4* are loaded when the second *loop* is executed. This frees at least 2 register which can in many cases give an increase in occupancy of up to 33%.

```
 1  kernel(ptr1,ptr2,ptr3,ptr4,ptr_result){
      ...
 3    for i=0 to N
        x += ptr1[i] * ptr2[i];
 5      y += ptr3[i] / ptr4[i];
      end
 7    ...
    }
```

```
    kernel(ptr1,ptr2,ptr3,ptr4,ptr_result){
 2    ...
      for i=0 to N
 4      x += ptr1[i] * ptr2[i];
      end
 6    for i=0 to N
        y += ptr3[i] / ptr4[i];
 8    end
      ...
10  }
```

Figure 1. (left): Pseudo code for a kernel that qualifies for loop *splitting*. (right): The same code after *loop splitting*.

2.2. Branch splitting

As *loop splitting*, the general idea behind *branch splitting* is to reduce the usage of hardware resources such as registers and shared memory of a kernel or at least part of the kernel. *Branch splitting* can be applied for any kernel that doesn't run with 100% occupancy, works on independent data and contains *branch*ing where the *branches* differ in complexity and therefore in the usage of hardware resources, especially registers or shared memory. This means that if one *branch* makes excessive usage of registers or shared memory so that the occupancy drops below 100%, the whole kernel will always run with that minimal occupancy even if the *branch* that leads to the lower occupancy is never executed.

The idea is to split the *branches* of the initial kernel into two kernels, where one kernel executes only the *if-branch* and the other kernel only executes the *else-branch*. The benefit of a two kernel version is that even we have a little overhead from the additional kernel invocation; we get an increase in performance since we could increase the occupancy for at least part of the initial kernel.

The worst case scenario for using the single kernel approach is when at least one *thread* per *warp* steps through another *branch* as the rest of the *thread*s, because of the SIMT architecture acts in such as way that in this case every *thread* of a *warp* has to step through the instructions of all *branches* and the device can only be utilized to the minimum occupancy defined by the *branch* with the highest usage of hardware resources. As an example we can see the figure 6, where the arithmetic calculations are chosen so that the *if-branch* uses fewer registers than the *else-branch*. In the split version shown in *Figure 2* the *if-kernel* uses an overall number of 6 registers compared to 13 for the *else-kernel*. This results in occupancy of 100% for the *if-branch* and 67% for the *else-branch*.

```
    branchedkenrnel(){
 2    load decision mask
      load input data used by both branches
 4    if decision mask[tid] == 0
        load input data for if branch
 6      perform calculations using 6 registers
      else if decision mask[tid] == 1
 8      load  input data for else branch
        perform calculations using 13 register
10    end if
    }
```

```
 1  ifkernel(){
      load decision mask
 3    if decision mask[tid] == 0
        load all input data
 5      perform calculations using 6 registers
      end if
 7  }

 9  elsekernel(){
      load decision mask
11    if decision mask[tid] == 1
        load all input data
13      perform calculations using 13 register
      end if
15  }
```

Figure 2. (left): Pseudo codes for the single kernel version. (right): the split/two-kernel version

2.3 Theoretical Analysis of Branch and Loop splitting

Experiments as the benchmark discussed in the next section have shown that this transformation in many cases can drastically improve performance. To get an idea of the theoretical speedup for the worst case the following formula can be used (Ec. 1):

$$speedup = \frac{T}{\sum_{i=1}^{n} \frac{t_i \times \rho_{min}}{\rho i} + \sigma} \quad (1)$$

Where *T*, is defined as the runtime for the worst case of the *branch*-version when the instructions of all *n branches* are executed. In ideal conditions, neglecting all optimizations that are applied at hardware level, this *T* can roughly be expected to be $T = \sum_{i=1}^{n} t_i \times \rho_i$.

Where ρ_i is defined as the occupancy for *i-th branch* when it run on its own ρ_{min}, is the occupancy when the *branch*ed version gets executed, σ is the invocation overhead produced every time a kernel is called, and t_i is the runtime of the single *branch* before the *splitting*.

The calculated speedup just gives an idea of what theoretic speedup can be expected if the kernel doesn't get limited by other factors e.g. the memory bank conflicts. There are some more factors that might reduce the speedup or prevent this transformation of being applied. As said before the kernel in its original setup might already have saturated the memory bandwidth where maybe the increased occupancy might help to hide part of the memory latency but as an overall the performance increase for this case might be marginal. The runtime of the single *branches* also plays a major roll, if the kernel that might run with 100% occupancy has a runtime that is much lower than the kernel running with 67%, then the additional occupancy might not outperform the overhead added. As a guideline we can consider the following conditions for a kernel to be considered for a *branch splitting*:

- A kernel that does not run at 100% occupancy
- A kernel that contains two or more major *branches*
- A kernel where the *branches* utilizing a different amount of hardware resources and the *branches* easily can be separated

3. Benchmark

In this section we demonstrate our optimizations on a complex parallel algorithm, the Lattice Boltzmann Method (LBM) that is already optimized for NVIDIA CUDA. First we present a brief introduction about the LBM, and then we show the application of our method to increases the performance of the algorithm.

3.1. Lattice Boltzmann Method

The Lattice Boltzmann Method (LBM) models Boltzmann particle dynamics on a 2D or 3D lattice [3], [4]. It is a microscopically inspired method designed to solve macroscopic fluid dynamics problems. It lives at the interface between the microscopic (molecular) and macroscopic (continuum) worlds. The Boltzmann equation expresses the variation of the average number of microscopic particles moving with a given velocity between each pair of neighboring sites. Such variation is caused by inter-particle interactions and ballistic motion of the particles. The variables associated with each lattice site are the particle distributions that represent the probability of particle presence with a given velocity. Particles stream synchronously along links from each site to its neighbors in discrete time steps and perform collision between consecutive streaming steps. The LBM is second-order accurate both in time and space, and in the limit of zero time step and lattice spacing, it yields the Navier-Stokes equations for an incompressible fluid [5], [6]. The time dependent movement of fluid particles at each lattice node satisfies the following particle propagation equation 2:

$$F_i(\mathbf{x} + \boldsymbol{e}_i, t+1) = F_i^{eq}(\mathbf{x}, \tau) - \frac{1}{t}[F_i(\mathbf{x}, t) - F_i^{eq}(\mathbf{x}, \tau)] \quad (2)$$

Where F_i is the non-equilibrium distribution function, F_i^{eq}=($\mathbf{x}$,τ) is the equilibrium distribution function, and $\mathbf{e}_i$ is the microscopic velocity at lattice node $\mathbf{x}$ at time t, respectively, and τ is the relaxation time which is a function of fluid viscosity. The subscript i represents the lattice directions around the node as shown in *figure 3* [3], [4].

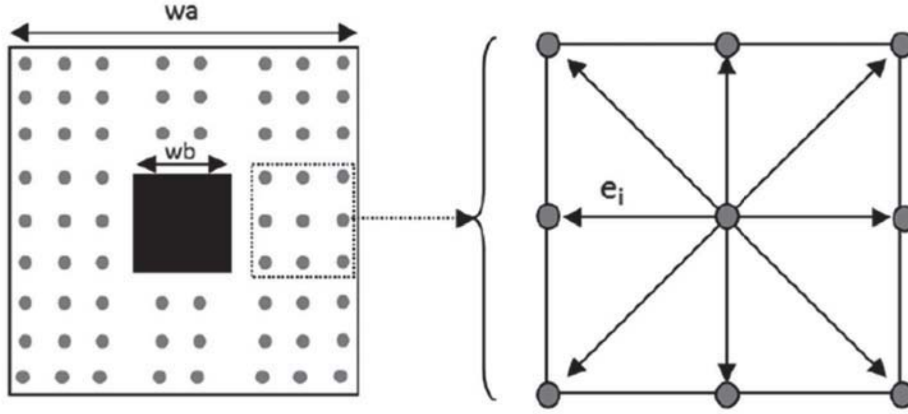

Figure 3. Particles advected based on *D2Q9* LBM model

The function is given in the following form for the two-dimensional model with nine microscopic velocity vectors (*D2Q9*) and three dimensional models with nineteen microscopic velocity vectors (*D3Q19*) (Ec. 3):

$$F_i^{eq} = w_i \rho \left[\frac{1}{3}(\boldsymbol{e}_i \times \boldsymbol{u}) + \frac{9}{2}(\boldsymbol{e}_i \times \boldsymbol{u})^2 - \frac{3}{2}(\boldsymbol{u} \times \boldsymbol{u})\right] \quad (3)$$

Where ρ is the density of the node, and w_i is the weight factor in the *i-th* direction. The weight factors (w_i) for the *D2Q9* LBM model are: $w_0 = \frac{16}{36}$ for rest particle, $w_1 = \frac{4}{36}$ for particles streaming to the face connected neighbors and $w_0 = \frac{1}{36}$ for particles streaming to the edge-connected neighbors as shown in *figure 3*. The weight factors are derived based on the lattice type (*DxQy*) and the derivations can be found in [3].

The macroscopic properties, density (ρ), momentum (U), and velocity (u) of the nodes are calculated using the following relations (Ec. 4, Ec. 5, Ec 6):

$$\rho = \sum_{i=1}^{Q} F_i \quad (4)$$

$$\mathbf{U} = \sum_{i=1}^{Q} Fi \times \boldsymbol{ei} \quad (5)$$

$$\mathbf{u} = \frac{\mathbf{U}}{\rho} \quad (6)$$

3.2. LBM Benchmark: A branch and Loop splitting approach

The implementation of our LBM algorithm has two computational intensive kernel, *Figure 4* and

Figure 5 show the pseudo code for both kernel; the first one called *boundary_kernel has to allocate* the boundary values of the arrows and columns for each particles, and the second one called *velocities_and_densities_kernel* is in charge to calculate the velocity and density values for each particle inside the system.

```
  non_split_loop_boundary_kernel(){
2   load geometry
    load input data
4   if geometry[tid] == solid boundary
      for(each particle on the boundary)
6       work on the boundary rows
        work on the boundary columns
8       store result
  }
```

Figure 4. Pseudo code for the non-split loop kernel version used in the LBM benchmark to allocate the boundary values of the arrows and columns for each particle

```
 1 branch_velocities_densities_kernel(){
     load geometry
 3   load input data
     if particles
 5     load temporal data
       for(each particle)
 7       if geometry[tid] == solid boundary
           load temporal data
 9         work on boundary
           store result
11       else
           load temporal data
13         work on fluid
           store result
15 }
```

Figure 5. Pseudo code for the non split *branch* kernel version used in the LBM benchmark to calculate the velocity and density values for each particle

Because of to the nature and the geometry of the problem we need to identify each particle before we can calculate its value, for this application we work with two different kinds of particles: fluid and solid boundaries. For that reason both kernels we were using implement flow control instructions, *if* and *for loop* statements, which allows to differentiate the particles, but at the same time can significantly impact the effective instruction throughput by causing *thread*s of the same *warp* to diverge[3].

To obtain best performance, and in order to minimize the number of divergent *warp*s and reduce the number of register used in the *non-split* kernel, we took in advantage that the first kernel was working on sets the solid boundary conditions of the system for both row and columns, hence, as is shown in *figure 6*; we split the *loop* statement into two *loop* statements, one for the rows and one for the columns. the reason to do that is because when we have the four array (two for the rows and two for the columns) inside the *loop*, we need to have four register dedicated to keep the value of the base address for each array, but when we split the for *loop* we just need two register (on for the columns and one for the rows) per *loop* instead of 4 register for one *loop*.

```
1  split_loop_boundary_kernel(){
     load geometry
3    load input data
     if geometry[tid] == solid boundary
5      for(each particle on the boundary)
         work on the boundary rows
7        store result
       for(each particle on the boundary)
9        work on the boundary columns
         store result
11 }
```

Figure 6. Pseudo code for the split *loop* kernel version used in the LBM benchmark to allocate the boundary values using one *loop* statement for the arrows, and another *loop* statement for the columns

Following the same idea, we applied a *branch splitting* on the second kernel trying to reduce the number of register. Taking in advances that *branch velocities densities kernel* is working with solid or fluid particles, and the number of register used inside the kernel is more than 10, which leads not to get 100% occupancy. The *figure 7* shows the split version of the *branch velocities densities kernel* which help to obtain a 100% occupancy for each kernel.

```
1  if_velocities_densities_kernel(){
     load geometry
3    load input data
     if particles
5      load temporal data
       for(each particle)
7        if geometry[tid] == boundary
           load  temporal data
9          work on boundary
           store result
11 }

13 else_velocities_densities_kernel(){
     load geometry
15   load input data
     if particles
17     load temporal data
       for(each particle)
19       if geometry[tid] == fluid
           load  temporal data
21         work on fluid
           store result
23 }
```

Figure 7. Pseudo code for the split *branch* kernel version used in the LBM benchmark to calculate the velocity and density values for each particle

Our results show that these techniques, depending on the problem layout and the algorithm, can lead to increase the occupancy and a drastic improvement in performance compared to *non-split* version of the algorithm. To Apply the *branch* and *loop splitting* method to the LBM problem shows that the *non-split* version algorithm take more time to compute the velocities and densities compared with the *split* version, and due to the nature of the problem, as much as we decrease the ratio of the system, which is defined as *wb*, the algorithm attempts to increase the number of step in order to converge to the *wa* results, leading in a speed up for the computation of the algorithm because of the use of specialized kernel for each *branch* of the code. In *figure 8*, for a 128x128 problem la-

yout, we can see that the split version increase the performance of the application in 30.1%, and for a 256x256 problem layout the performance increase approximately 24% .

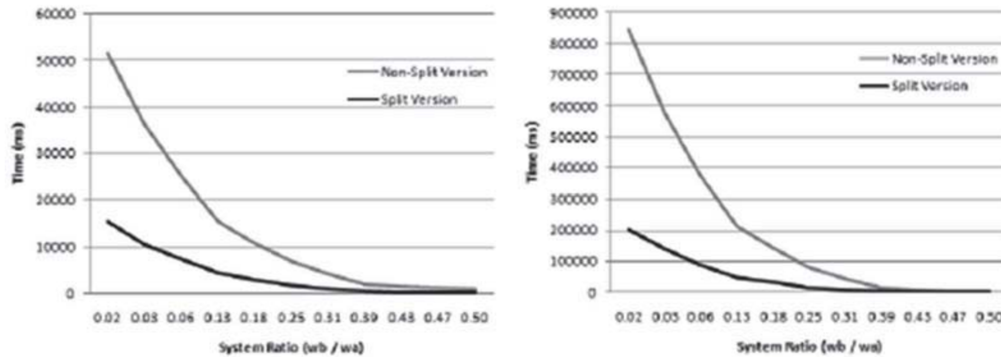

Figure 8. Time per system ratio of a *D2Q9* LMB simulation (left): for a 128x128 problem layout. (right) for a 256x256 problem layout.

4. Conclusion

To run kernels with the highest possible occupancy is one of the major tasks for any GPGPU programmer. Any transformation or optimization that can reduce the usage of hardware resources that reduce the occupancy are a big contribution to the overall performance of a GPGPU program execution. Our *loop* and *branch* transformation helps to increase the occupancy and parallelism for some special cases of *loop*s and a more general case of *branches*.

The *Loop splitting* is an example for a transformation that might seem counterproductive on most other architectures than a GPU, but here where occupancy is a major player in the performance game, it can have a positive impact on the overall performance.

In any case *branches* are not a good thing to use in any SIMD or SIMT architecture, but for some algorithms there are not that many other efficient ways to implement them without using *branch*ing. Therefore optimizing *branches* in a way that either the number of instructions per *branch* gets minimized or the input data set reduces the probability of *warp*s where both *branches* have to be executed is a major task for GPGPU programmers. In many cases there is no way to prevent both *branches* from being executed within a *warp*. Furthermore in many cases the *branches* differ in complexity and therefore in the usage of hardware resources. In those cases *branch splitting* is a promising transformation that can drastically improve the performance of a GPGPU application.

References

[1] NVIDIA. *Compute Unified Device Architecture Programming Guide*. NVIDIA: Santa Clara, CA, 2007.

[2] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Wen-Mei. *Optimization principles and application per-*

formance evaluation of a multithreaded GPU using CUDA. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 73–82, 2008.

[3] X. He And L. Luo. *Lattice Boltzmann Model for the Incompressible Navier–Stokes Equation*. Journal of Statistical Physics, 88(3):927–944, 1997.

[4] M. Kutay, A. Aydilek, and E. Masad. *Laboratory validation of lattice Boltzmann method for modeling pore-scale flow in granular materials*. Computers and Geotechnics, 33(8):381–395, 2006.

[5] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. *NVIDIA Tesla: A Unified Graphics and Computing Architecture*. IEEE Micro, pages 39–55, 2008.

[6] Y. ZHAO. *Lattice Boltzmann based PDE solver on the GPU*. The Visual Computer, 24(5):323–333, 2008.