CUDA parallelization and performance assessment of the Sciara-fv2 Cellular Automata lava-flows simulation model

Antonino Natale*[1] and Matteo Perfidio**[1]

[1]Department of Mathematics and Computer Science, University of Calabria, Rende, Italy

February 2022

Abstract

In this paper we presents a CUDA parallelization of the deterministic Macroscopic Cellular Automata model SCIARA for simulating lava flows. Furthermore, we present the results of an experimental analysis we conducted in order to evaluate performance-wise impact. Specifically, given benchmarks of our parallel implementation with respect to serial algorithm, we observe how parallel implementation is very competitive and even faster.

Keywords: Macroscopic Cellular Automata, Numerical Simulation, Modelling, Lava Flows, Mt Etna, Anisotropy Proble, CUDA, Parallel Computing.

1 Introduction

Fluid dynamics is a notoriously difficult branch of physics, where very few problems can be solved analytically. A fluid can undergo linear oscillations just like a solid, but unlike solid can also flow around obstacles and, in many circumstances also rotate swirling on itself. This behavior is inherently nonlinear and difficult to quantify. Normally, we describe the macroscopic state of a fluid in terms of density ρ, and velocity $\vec{u}$, both as a function of space

*ntlnnn97r06e041t@studenti.unical.it
**prfmtt98e07f537p@studenti.unical.it

and time. These functions are followed by a set of partial differential equations, called *Navier-Strokes equations*, which derive from Newton's laws and some empirical assumptions about fluid behavior. Because of the analytical complexity of the Navier-Strokes equations, in fluid dynamics problems, a computational approach is preferred. Thus, the *Computational Fluid Dynamics (CFD)*, methods of discretization of the Navier-Strokes equations, resulting in numerical resolution for given boundary conditions.

Cellular Automata

Cellular Automata (CA) are parallel computing models, widely used for modelling and simulating complex systems, whose evolution can be described in terms of local interactions. However, major interest for CA regard their use in Complex Systems modelling in various fields like Physics, Biology, Earth Sciences and Engineering. Besides theoretical studies, CA have been applied to a variety of fields such as pattern recognition, image processing and cryptography. In particular, in this paper we introduce a CA regards lava flows CFD simulation. Lava flows can be considered one of the most dangerous and difficult phenomena to be modelled as, for instance, temperature drops along the path by locally modifying the magma dynamical behaviour. Lava flows generally evolve on complex topographies that can change during eruptions, due to lava solidification, and are often characterised by branching and rejoining of the flow. Nevertheless, many attempts of modelling real cases can be found in literature. One of these adopts CA for modelling Etnean lava flows (Italy) through the numerical simulation code SCIARA. In this paper, we present a new parallel implementation of the latest CA model SCIARA-fv2 release, which differs with respect to [1] as it introduces GPU-aided parallel computation through CUDA API. In the end, the model we proveded can be able to reproduce Etna lava flows in an extremely short time compared to the original version.

Macroscopic Cellular Automata

Classical homogeneous Cellular Automata can be viewed as an n-dimensional space, subdivided in cells of uniforms shape and size. Each cell embeds an identical finite automaton (*fa*), whose state accounts for the temporary features of the cell; Q is the finite set of states. The *fa* input is given by the states of a set of neighbouring cells, including the central cell itself. The neighbourhood conditions are determined by a geometrical pattern, X, which is invariant in time and space. The *fa* have an identical state tran-

sition function τ:$Q^{\#X} \rightarrow Q$, which is simultaneously applied to each cell. At step t=0, *fa* are in arbitrary states and the CA evolves by changing the state of all *fa* simultaneously at discrete times, according to τ. Moreover, a formal explaination can be found in [1].

2 SCIARA-fv2 Model

In formal terms, SCIARA is defined as:

$$\text{SCIARA} = < R, L, X, Q, P, \tau, \gamma >$$

where:

- R is the set of square cells covering the bi-dimensional finite region where the phenomenon evolves;
- $L \in R$ specifies the lava source cells (i.e. craters);
- $X = (0,0), (0,1), (-1,0), (1,0), (0,-1), (-1,1), (-1,-1), (1,-1), (1,1)$ identifies the pattern of cells (Moore neighbourhood) that influence the cell state change; in the following we will refer to cells by indexes 0 (for the central cell) through 8;
- $Q = Q_z \times Q_h \times Q_T \times Qf_f^8$ is the finite set of states, considered as Cartesian product of "substates". Their meanings are: cell altitude a.s.l., cell lava thickness, cell lava temperature, and lava thickness outflows (from the central cell toward the four adjacent cells), respectively;
- $P = \{w, t, T_{sol}, T_{vent}, r_{T_{sol}}, r_{T_{vent}}, hc_{T_{sol}}, hc_{T_{vent}}, \delta, \rho, \epsilon, \sigma, c_v\}$ is the finite set of parameters (invariant in time and space) which affect the transition function;
- $\tau : Q^7 \rightarrow Q$ is the cell deterministic transition function; it is outlined in the following sections;
- $\gamma : Q_h \times N \rightarrow Q_h$ specifies the emitted lava thickness from the source cells at each step $k \in N$ (N is the set of natural numbers).

3 Implementation

The parallel implementation of the Sciara Cellular Automa was developed with the support of CUDA API. CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). CUDA is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

3.1 Algorithm

In the reference paper [1], the state variables (substates), are Q_z, Q_h, Q_T, and Q_f^8. In the serial implementation, the first three substates are called `Sz`, `Sh` and `ST`, and are defined as double-precision linear arrays representing two-dimensional matrices, that cover the whole cellular space (or domain) R. Accordingly to this choice, the domain `R` is implicitly considered in the substates. The `Sz_next`, `Sh_next` and `ST_next` support arrays are also defined, which are used to write the new state values for the cell, as computed by the transition function's elementary processes (i.e., the kernels). The outflows from the central cell toward the neighbors, which are defined as Q_f^8, are implemented as a single buffered structure made by 8 linear arrays, one for each adjacent cell, and is called `Mf`. The X_i and X_j arrays therefore correspond to the X geometrical pattern of the formal definition of Sciara-fv2. Therefore, it is sufficient to use the $(i + X_i[k], j + X_j[k])$ coordinates to access the k^th neighboring cell of the cell (i, j).

We introduce now a formal declaration of mentioned above algorithm.

Algorithm 1: Sciara-fv2 serial

Input: Let S_h, S_T, S_z state buffers, $S_{h_{next}}$, $S_{T_{next}}$, $S_{z_{next}}$ the respective transitions buffers, τ the elapsed time and σ the current iteration step, then computes Etnee lava flows simulation described in section 2 for the current step $0 \leq \sigma \leq 16000$.

Result: Let Ψ the total current lava for each iteration step σ.

```
1  function Step():
2      τ := τ + P_clock
3      σ := σ + 1
4      foreach r ∈ cell do
5          emitLava(r, ...)
6      end
7      S_h ← S_hnext
8      S_T ← S_Tnext
9      foreach r ∈ cell do
10         computeOutflows(r, ...)
11     end
12     foreach r ∈ cell do
13         massBalance(r, ...)
14     end
15     S_h ← S_hnext
16     S_T ← S_tnext
17     foreach r ∈ cell do
18         computeNewTemperatureAndSolidification(r, ...)
19     end
20     S_z ← S_znext
21     S_h ← S_hnext
22     S_T ← S_Tnext
23     foreach r ∈ cell do
24         boundaryConditions(r, ...)
25     end
26     S_h ← S_hnext
27     S_T ← S_Tnext
28     let Ψ := reduceAdd(S_h, ...)
29     return Ψ
30     end
```

3.2 Basic concepts

Before being able to provide a formal description of the parallel algorithm, it is necessary to make some clarifications. The working context, where the algorithm operates, has some peculiarities which, in order to understand it, we will introduce in the next paragraph. Before that, we introduce a description image of CUDA Memory Model.

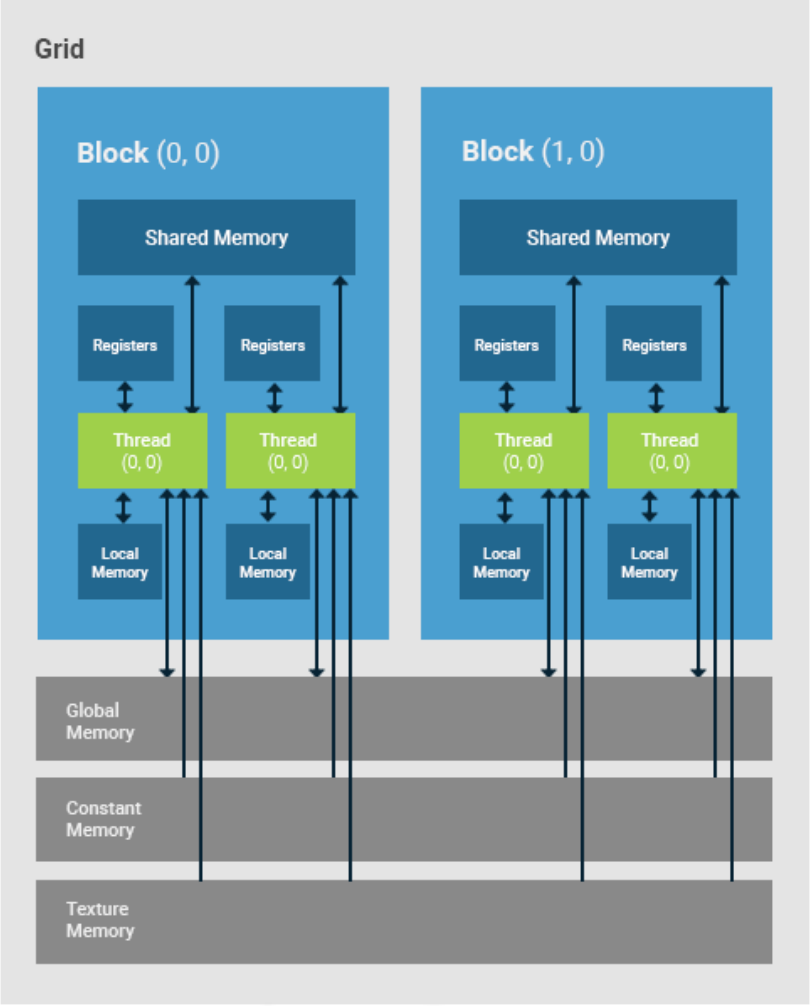

Figure 1: CUDA Memory Model

3.2.1 Constant Memory

Constant memory [6] is a read-only cache which content can be broadcasted to multiple threads in a block. A variable allocated in constant memory needs to be declared in CUDA by using the special `__constant__` identifier, and it must be a global variable, i.e. it must be declared in the scope that contains the kernel, not inside the kernel itself. [6]

```
39
40  __constant__ int Xi[MOORE_NEIGHBORS];
41  __constant__ int Xj[MOORE_NEIGHBORS];
42  __constant__ TVent vents[VENTS_COUNT];
43
```

Figure 2: Use of Constant Memory on 'cuda-tiled-halo.cu'

3.2.2 Shared Memory

Shared memory [6] is on-chip and is much faster than local and global memory. Shared memory latency is roughly 100x lower than uncached global memory latency. Threads can access data in shared memory loaded from global memory by other threads within the same thread block. Memory access can be controlled by thread synchronization to avoid race condition `__syncthreads()`. Shared memory can be uses as user-managed data caches and high parallel data reductions. Shared memory are accessable by multiple threads.

```
100
101     extern __shared__ double shared[];
102
103     double* sh_shared = &shared[0];
104     double* sz_shared = &shared[TILE_PITCH * TILE_PITCH];
105
```

Figure 3: Use of Shared Memory on 'cuda-tiled-halo.cu'

3.2.3 Unified Memory

Unified Memory [5] is a single memory address space accessible from any processor in a system. This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs. Allocating Unified Memory is as simple as replacing calls to `malloc()` or new with calls to `cudaMallocManaged()`, an allocation function that returns a pointer accessible from any processor.

```
8
9       cudaMallocManaged(&Sz, M * N * sizeof(double));
10
```

Figure 4: Allocate Unified Memory on 'cuda-straightforward.cu'

3.2.4 Warp-Level Primitives

NVIDIA GPUs and the CUDA programming model employ an execution model called SIMT (Single Instruction, Multiple Thread). NVIDIA GPUs execute warps of 32 parallel threads using SIMT, which enables each thread to access its own registers, to load and store from divergent addresses, and to follow divergent control flow paths. We introduce Warp-Level Primitives [3] in `reduceAdd` kernel to achieve even higher performance by using explicit warp-level programming.

```
447     if(threadIdx.x < 32) {
448         v += __shfl_down_sync(0xFFFFFFFF, v, 32);
449         v += __shfl_down_sync(0xFFFFFFFF, v, 16);
450         v += __shfl_down_sync(0xFFFFFFFF, v,  8);
451         v += __shfl_down_sync(0xFFFFFFFF, v,  4);
452         v += __shfl_down_sync(0xFFFFFFFF, v,  2);
453         v += __shfl_down_sync(0xFFFFFFFF, v,  1);
454     }
455
```

Figure 5: Use of Warp-Level Primitives on 'cuda-tiled-halo.cu'

3.3 Straightforward

The straightforward version introduces a several number of changes from the original Sciara-fv2 serial version in order to achieve a first naïve parallel implementation. The steps taken to develop this version are described below:

1. First, we rewrite all transition function in CUDA kernels.
2. We completely rewrite `emitLava` kernel to avoid a waste of unused threads, because the effective algorithm performs only n-times, where n is the count of vents.
3. We introduce `reduce_add` and `memcpy_gpu` to replace the serial slowly function.
4. X_i, X_j and *vents* was placed in the Constant Memory, described in Section 3.2.2, as they were read-only memory areas.
5. All state and transition buffers was allocated in the Unified Memory, described in Section 3.2.3.

3.3.1 Performance Assessment

In this section, we presents the tables containing the execution times of the straightforward version. All the time refers to whole simulation execution, including final processing result times and other stuff.

Type	Threads blocks	Time	Speed-Up
Straightforward	8x8	10.895 s	26.917
Straightforward	16x16	12.734 s	23.085
Straightforward	32x32	16.390 s	17.893

Table 1: Straightforward - Parallel executions times.

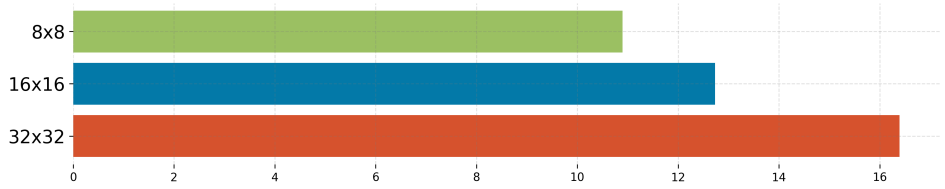

Figure 6: Absolute execution times of Straightforward Version

3.3.2 Roofline

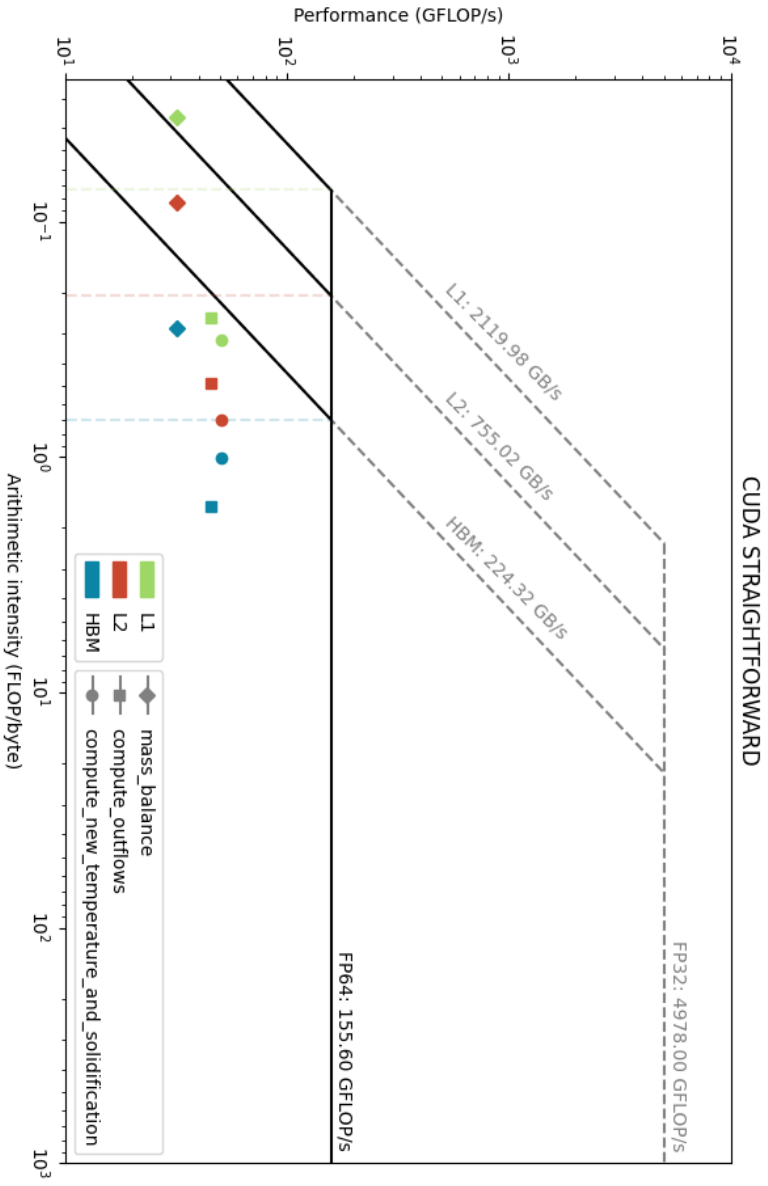

Figure 7: Roofline model of Straightforward Version

3.4 Tiled

CUDA defines registers, shared memory, and constant memory that can be accessed at a higher speed and in a more parallel manner than the global memory. Using these memories effectively will likely require redesign of the algorithm. A tiled algorithm would load all input elements for calculating all output elements into the Shared Memory per each block. The elements that are involved in multiple tiles are commonly referred to as halo cells or skirt cells. The size of this memory area is limited. Their capacities are implementation dependent. Once their capacities are exceeded, they become limiting factors for the number of threads that can be simultaneously executing in each SM.

The tiled versions of the Sciara-fv2 algorithm introduce several changes compared to the straghtforward version in order to obtain a consistent increase of performance. The steps taken to develop this version are described below:

1. All changes adopted from the straightforward version.
2. Use of Warp-level Primitives to optimize per-warp thread communication.
3. Introduction of Shared Memory and application of the tiled technique on kernels: `mass_balance`, `compute_overflows` and `reduce_add`.
4. Application of the grid-stride loop technique in order to perform multiple operations in a single thread on kernels: `memcpy_gpu` and `compute_new_temperature_and_solidification`.
5. Removal of transition states: Sh_{next}, ST_{next} and Sz_{next}, because the transition is computed locally per-block with the Shared Memory and thread synchronization.

3.4.1 Tiled with Halo

In the first variant of the tiled version, we introduced halo cells. Halo cells also referred to as skirt cells are all those cells involved in multiple tiles since they "hang" from the side of the part that is used solely by a single block. Furthermore, other changes have been applied:

1. Introduction of the Sz_{halo} buffer for sharing ghost cells.

2. Replacing the `memcpy_gpu` kernel (no longer needed after removing the transition buffers) with `prepare_halo` kernel. This last one loads only the shared cells between adjacent blocks instead the whole buffer.

3.4.2 Performance Assessment

In this section, we presents the tables containing the execution times of the Tiled with Halo version. All the time refers to whole simulation execution, including final processing result times and other stuff.

Type	Threads blocks	Time	Speed-Up
Tiled with Halo	8x8	5.386 s	54.450
Tiled with Halo	16x16	5.798 s	50.581
Tiled with Halo	32x32	8.625 s	34.002

Table 2: Tiled with Halo - Parallel executions times.

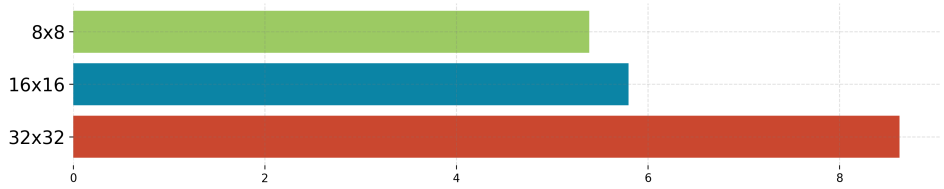

Figure 8: Absolute execution times of Tiled with Halo Version

3.4.3 Roofline

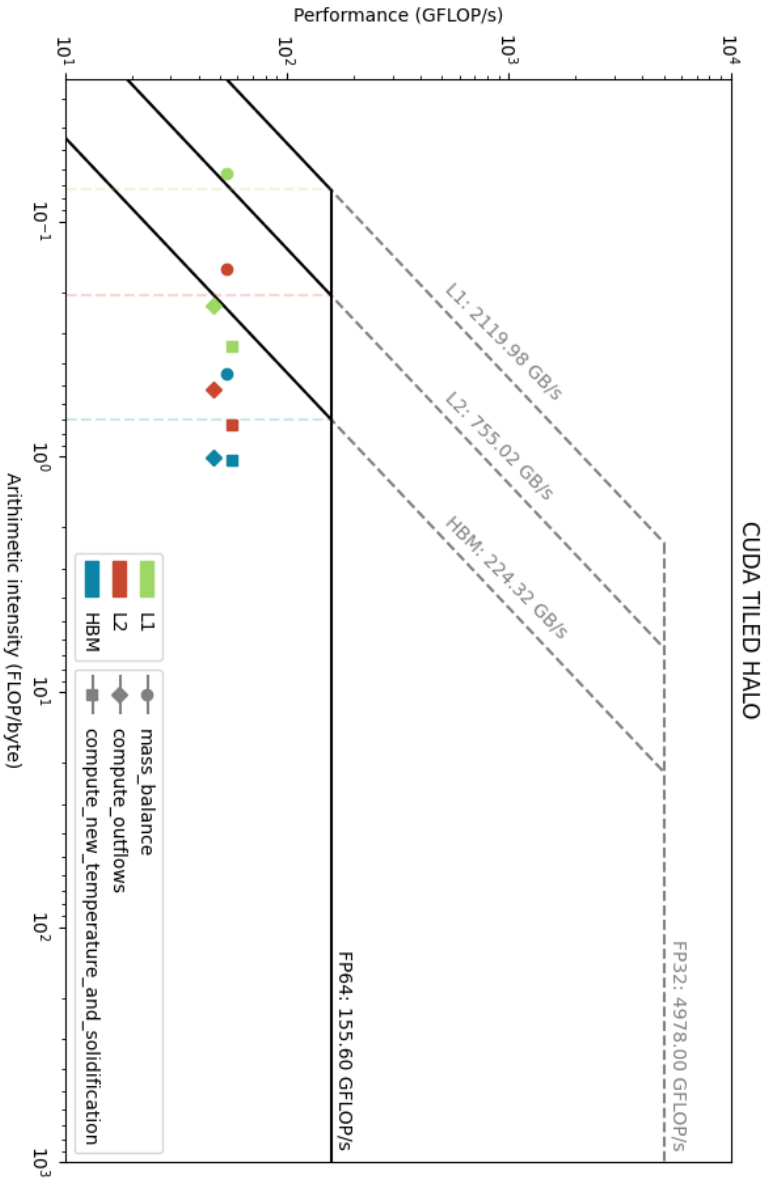

Figure 9: Roofline model of Tiled with Halo Version

3.4.4 Tiled without Halo

In the second variant of the tiled version, halo cells are no longer required by the instructions. For this reason, we had to reintroduce the Sz_{next} transition buffer.

3.4.5 Performance Assessment

In this section, we presents the tables containing the execution times of the Tiled without Halo version. All the time refers to whole simulation execution, including final processing result times and other stuff.

Type	Threads blocks	Time	Speed-Up
Tiled without Halo	8x8	5.097 s	57.537
Tiled without Halo	16x16	5.753 s	51.098
Tiled without Halo	32x32	8.770 s	33.440

Table 3: Tiled without Halo - Parallel executions times.

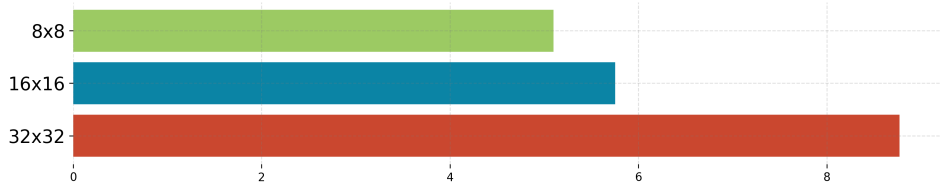

Figure 10: Absolute execution times of Tiled without Halo Version

3.4.6 Roofline

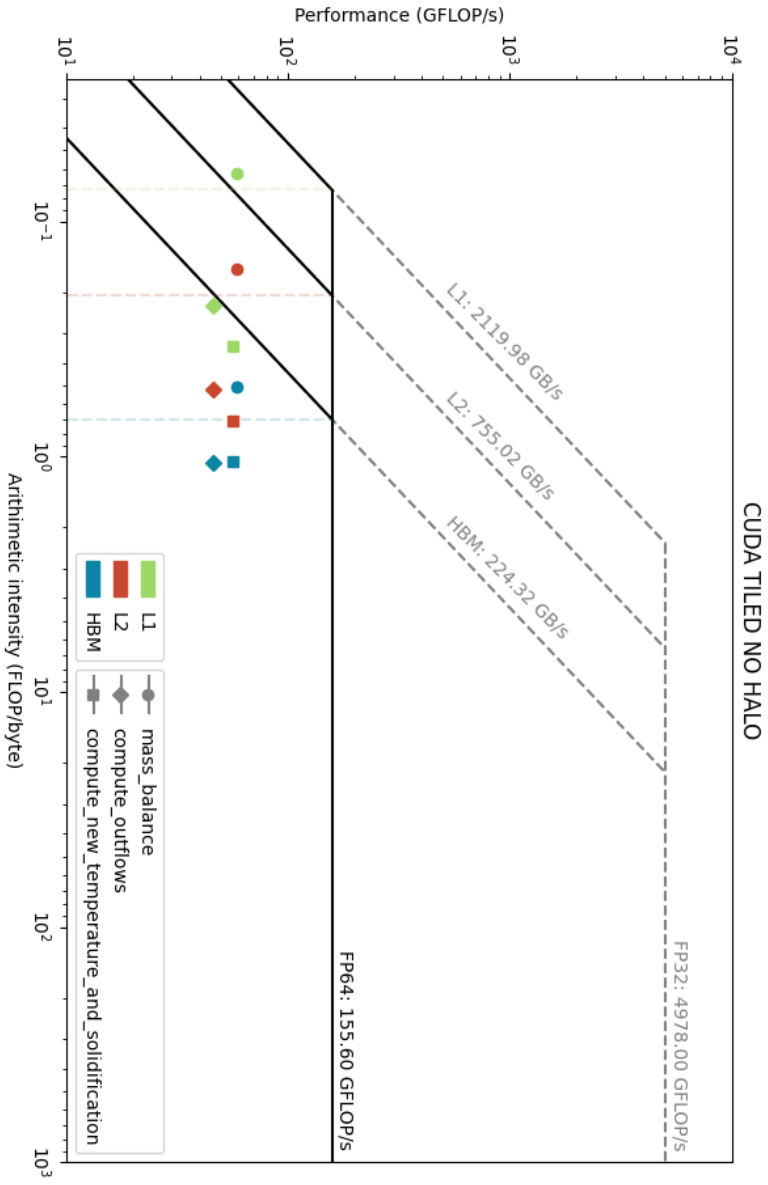

Figure 11: Roofline model of Tiled without Halo Version

3.5 Multi-GPU

In this version we introduce a new set of basic concept in order to implement a CUDA version for multiple GPU's devices.

3.5.1 Peer-to-Peer and Unified Virtual Addressing

Unified Virtual Addressing [2], or UVA, provides a single virtual memory address space for all memory in the system, and enables pointers to be accessed from GPU code no matter where in the system they reside, whether its device memory (on the same or a different GPU), host memory, or on-chip shared memory. Multiple GPU share data between each other using host memory with significant performance slow-down. In order to achieve high bandwidth and low latency communication between multiple GPU's CUDA introduces Peer-to-Peer Direct Transfers, where a GPU_0 initiates DMA transfers to GPU_1 and viceversa through PCIe NVLink.

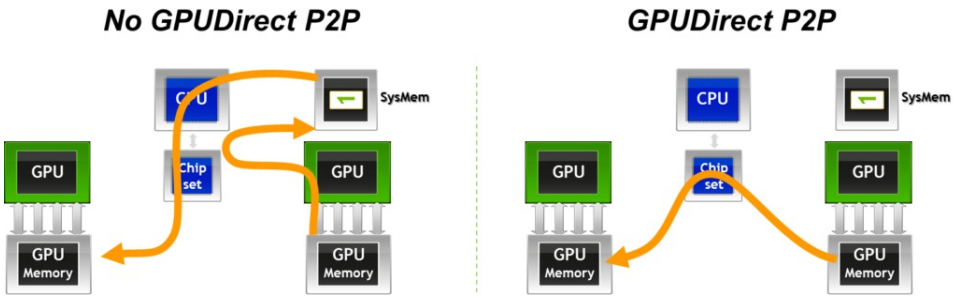

Figure 12: Peer-to-Peer vs. No-Peer-to-Peer

We introduce Peer-to-Peer and UVA capabilites with `cudaMemcpyDefault()`, `cudaMemcpyPeerAsync()` and `cudaDeviceEnablePeerAccess()` APIs.

3.5.2 OpenMP

OpenMP (Open Multi-Processing)[4] is an API that supports multi-platform shared-memory multiprocessing programming. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP is an implementation of multithreading, a method of parallelizing whereby a primary thread (a series of instructions

executed consecutively) forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

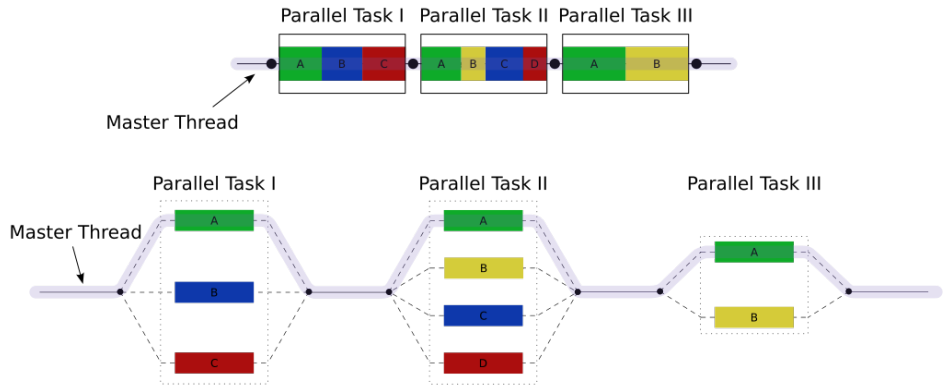

Figure 13: An example of OpenMP multithreading program

We introduce OpenMP to handle multiple GPU's management with `cudaSetDevice()` API per-thread for each GPU.

3.5.3 Implementation

The Multi-GPU version of the Sciara-fv2 algorithm introduce several changes compared to the Single-GPU's version in order to obtain a consistent increase of performance. The steps taken to develop this version are described below:

1. All changes adopted from the straightforward version.
2. All transitional buffers has been splitted for each GPU.
3. Rewriting of MF structure to improve transfer bandwidth between multiple GPU's.
4. Introduction of OpenMP APIs to perform multiple GPU's management.
5. Use of Warp-level Primitives to optimize per-warp thread communication.

6. Introduction of CUDA Peer-to-Peer transfers to deal with halo cells communication between GPU's.

7. Introduction of Shared Memory and application of the tiled technique on kernels: `mass_balance`, `compute_overflows` and `reduce_add`.

8. Application of the grid-stride loop technique in order to perform multiple operations in a single thread on kernels: `memcpy_gpu` and `compute_new_temperature_and_solidification`.

9. Removal of transition states: Sh_{next}, ST_{next}, because the transition is computed locally per-block with the Shared Memory and thread synchronization.

3.5.4 Performance Assessment

In this section, we presents the tables containing the execution times of the Multi-GPU version. All the time refers to whole simulation execution, including final processing result times and other stuff.

Type	Threads blocks	Time	Speed-Up
Multi-GPU	8x8	4.554 s	64.398
Multi-GPU	16x16	4.998 s	58.677
Multi-GPU	32x32	6.220 s	47.149

Table 4: Multi-GPU - Parallel executions times.

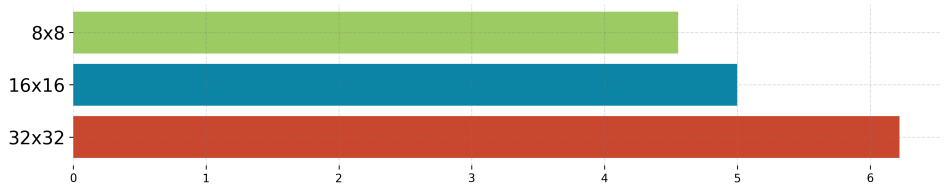

Figure 14: Absolute execution times of Multi-GPU Version

3.5.5 Roofline

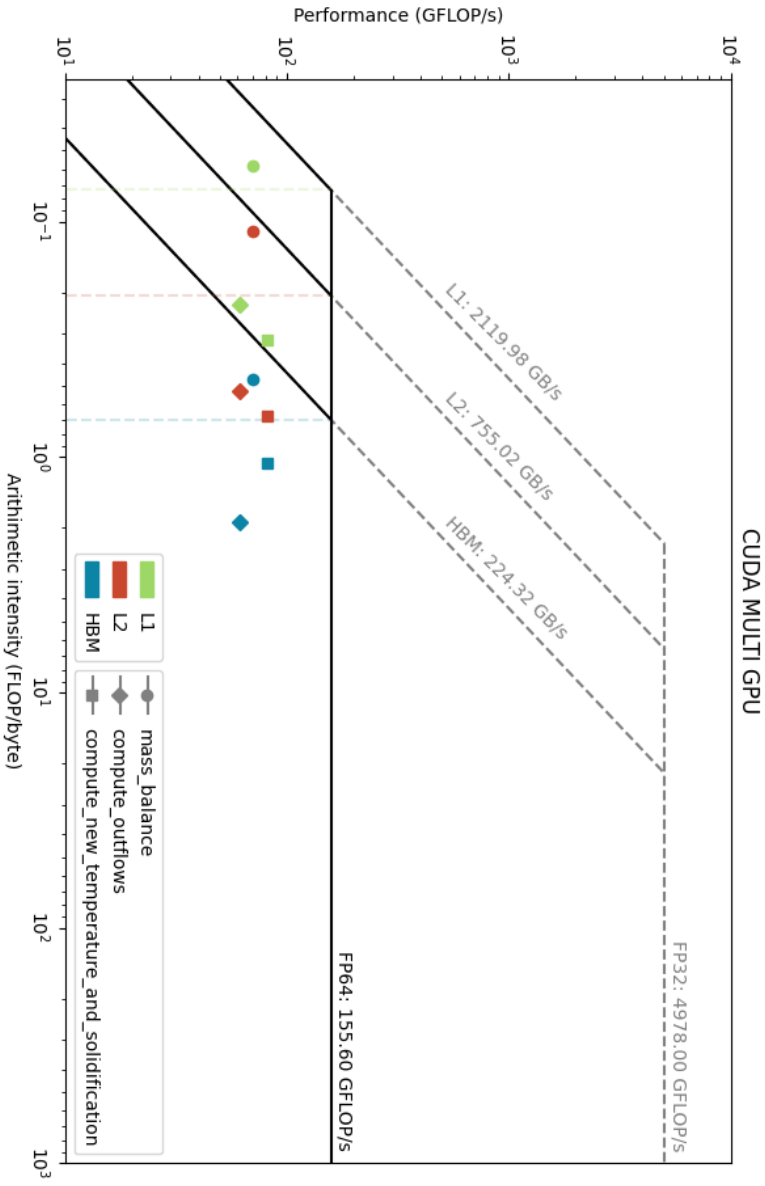

Figure 15: Roofline model of Multi-GPU Version

4 Grid Geometries

Let $\mathcal{V}$ a set of *vents*, $\mathcal{T}$ the number of thread per block, respectively 8, 16 or 32 as suggested by Cuda Occupancy Calculator and M and N the size of rows and columns of the simulation world matrix.

Kernel	Grid X	Grid Y	Shared Memory
emit_lava	$\lvert\mathcal{V}\rvert \div \mathcal{T}$	1	0
compute_outflows	$N \div \mathcal{T}$	$M \div \mathcal{T}$	$\mathcal{T}^2 \times 2$
mass_balance	$N \div \mathcal{T}$	$M \div \mathcal{T}$	$\mathcal{T}^2 \times 9$
compute_new_temp...	$N \div \mathcal{T}$	$M \div \mathcal{T}$	0
prepare_halo	$N \div \mathcal{T}$	$M \div \mathcal{T}$	0
reduce_add	$M \times N \div \mathcal{T} \div 8$	1	$\mathcal{T}^2$
memcpy_gpu	$N \times M \div \mathcal{T} \div 8$	1	0

Table 5: Sciara-fv2 Grid Geometries.

Please note: in Multi-GPU implementation each grid size has been divided for the number of GPU's.

5 Experiments

In this chapter, we present comparative analysis with the aim of evaluating the real performance of the CUDA parallel implementation of Sciara-fv2.

5.1 Configuration

All the experiments shown below were performed on an x86-64 machine, named JPDM2 with the following characteristics:

- Processor : Intel(R) Xeon(R) CPU E5440 @ 2.83GHz
- GPU : NVIDIA GeForce GTX 980 4 GB GDDR5 @ 1750MHz (×2)
- RAM : 16GB
- Kernel : Linux 5.14.15-arch1-1
- Compiler : GCC 11.1.0 / NVCC v10.2.89

6 Plots

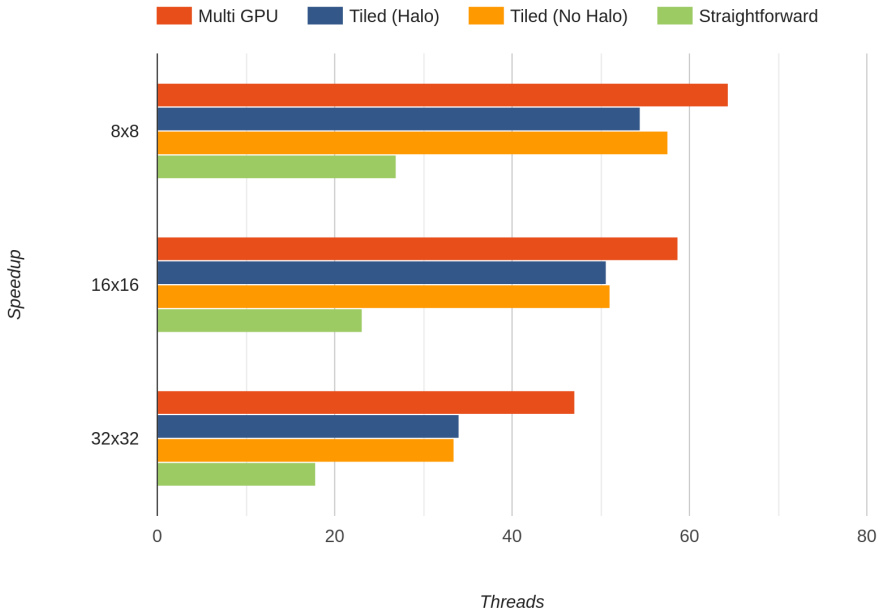

Figure 16: Parallel executions Speedup – 1

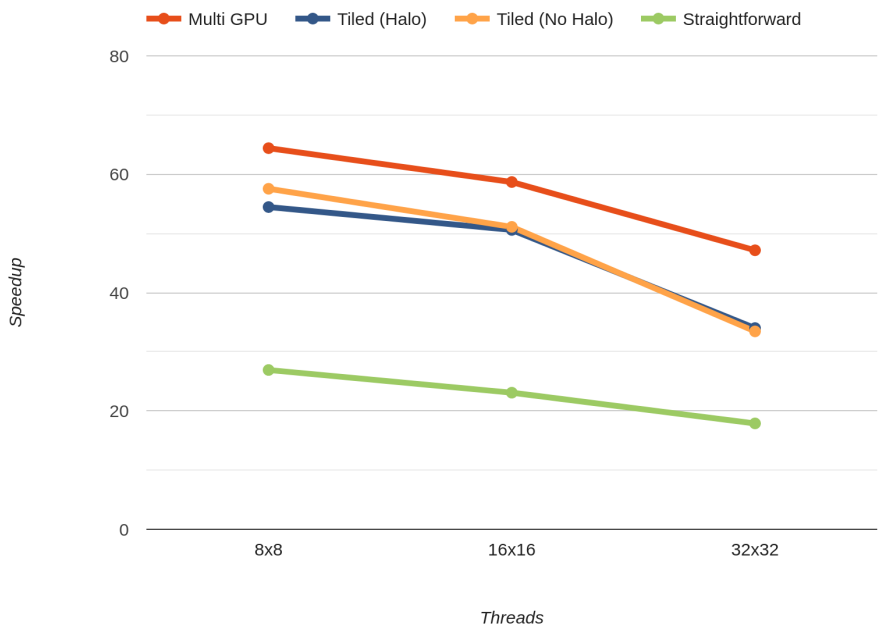

Figure 17: Parallel executions Speedup – 2

7 Tables

In this section, we presents the tables containing execution times and speed-up of all Sciara-fv2 parallel versions. All the time refers to whole simulation execution, including final processing result times and other stuff.

Version	Time
Serial	293.27 s
OpenMP	151.84 s

Table 6: Mean time of serial executions.

Please note: The following time has been obtained by compiling serial and OpenMP versions with 'level 3' optimizations and OpenMP was run with 4 threads.

Type	Threads blocks	Time	Speed-Up
Straightforward	8x8	10.895 s	26.917
Straightforward	16x16	12.734 s	23.085
Straightforward	32x32	16.390 s	17.893
Tiled with Halo	8x8	5.386 s	54.450
Tiled with Halo	16x16	5.798 s	50.581
Tiled with Halo	32x32	8.625 s	34.002
Tiled without Halo	8x8	5.097 s	57.537
Tiled without Halo	16x16	5.753 s	51.098
Tiled without Halo	32x32	8.770 s	33.440
Multi-GPU	8x8	4.554 s	64.398
Multi-GPU	16x16	4.998 s	58.677
Multi-GPU	32x32	6.220 s	47.149

Table 7: Mean time of parallel executions.

Please note: The following time has been obtained by compiling all CUDA versions with 'level 3' optimizations.

8 Results

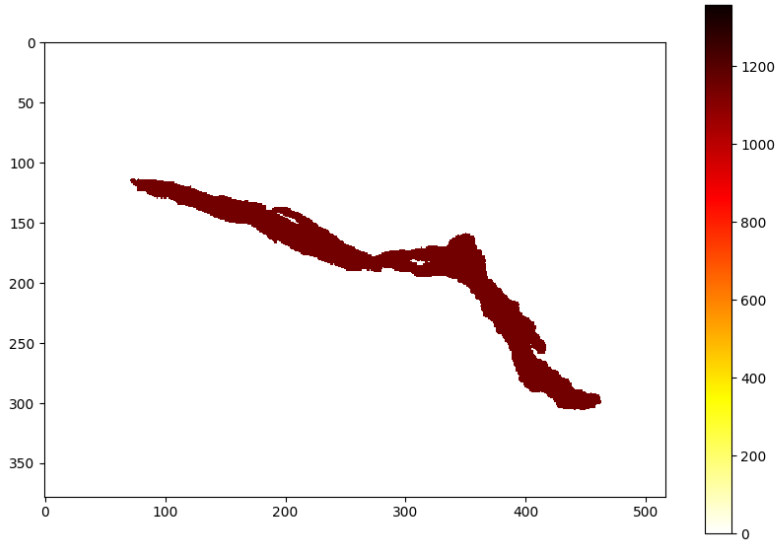

Figure 18: Temperature

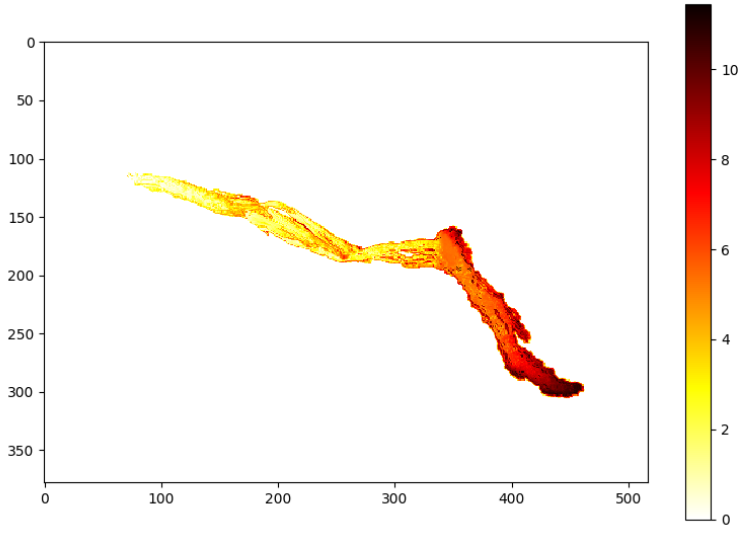

Figure 19: Solidified Lava Thickness

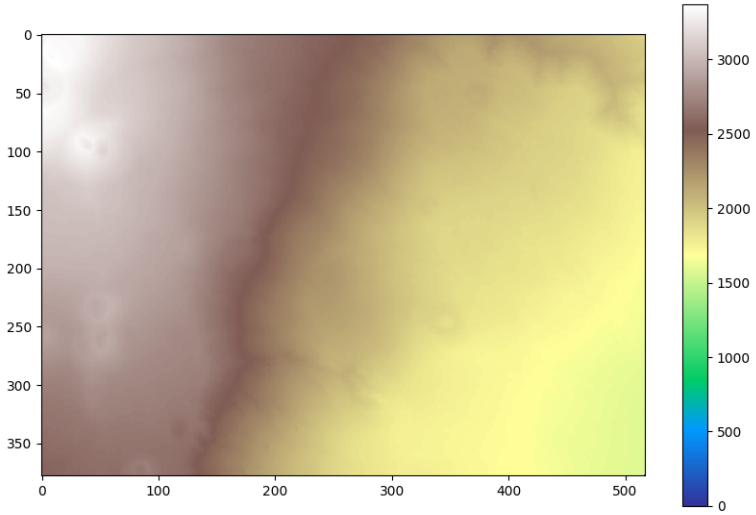

Figure 20: Morphology

9 Conclusion

The activities carried out within this paper focus on the study and implementation of a series of parallel versions via CUDA of the Sciara-fv2 Cellular Automaton. In particular, four parallel versions have been created, namely: Straightforward, Tiled with Halo, Tiled without Halo and Multi-GPU. In this regard, it was necessary to study techniques and strategies in order to increase the general performance of the serial algorithm. In particular, concepts such as Unified Memory Addressing (UVA) and Peer-to-Peer Direct Transfers have been deepened and used in the Multi-GPU version that has proven to be the most powerful version, reaching a speed-up definitely in line, if not beyond, the initial expectations. The speed-up obtained in the various implementations could be further improved in future versions by restyling the transition buffers inherited from the serial version. The current access pattern in fact creates bank conflict, as the accesses are purely non-coalesced.

List of Figures

List of Tables

References

[1] W. Spataro, M.V. Avolio, V. Lupiano, G.A. Trunfio, R. Rongo, and D. D'Ambrosio. The latest release of the lava ows simulation model SCIARA: first application to Mt Etna (Italy) and solution of the anisotropic flow direction problem on an ideal surface. In Procedia Computer Science, volume 1, pages 17-26, 2010.

[2] Schroeder, Tim C. "Peer-to-peer & unified virtual addressing." GPU Technology Conference, NVIDIA. 2011.

[3] De Gonzalo, Simon Garcia, et al. "Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs." 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2019.

[4] Chandra, Rohit, et al. Parallel programming in OpenMP. Morgan kaufmann, 2001.

[5] Landaverde, Raphael, et al. "An investigation of unified memory access performance in CUDA." 2014 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2014.

[6] Su, Ching-Lung, et al. "Overview and comparison of OpenCL and CUDA technology for GPGPU." 2012 IEEE Asia Pacific Conference on Circuits and Systems. IEEE, 2012.