GPU-CC: a Reconfigurable GPU Architecture with Communicating Cores

Gert-Jan van den Braak Henk Corporaal
Dept. of Electrical Engineering, Electronic Systems Group
Eindhoven University of Technology, The Netherlands
{g.j.w.v.d.braak, h.corporaal}@tue.nl

ABSTRACT

GPUs have evolved to programmable, energy efficient compute accelerators for massively parallel applications. Still, compute power is lost in many applications because of cycles spent on data movement and control instead of computations on actual data. Additional cycles can be lost as well on pipeline stalls due to long latency operations.

To improve performance and energy efficiency, we introduce GPU-CC: a reconfigurable GPU architecture with communicating cores. It is based on a contemporary GPU, which can still be used as such, but also has the ability to reorganize the cores of a GPU in a reconfigurable network. In GPU-CC data movement and control is implicit in the configuration of the communication network. Additionally each core executes a fixed instruction, reducing instruction decode count and increasing energy efficiency. We show a large performance potential for GPU-CC, e.g. 1.9× and 2.4× for a 3×3 and 5×5 convolution application. The hardware cost of GPU-CC is mainly determined by the buffers in the added network, which amounts to 12.4% of extra memory space.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Design, Performance

Keywords

GPGPU, reconfigurable architecture

1. INTRODUCTION

Single core performance growth halted in 2004 [2] with processors reaching their power consumption limit. Processors became multi-cores and GPUs (Graphics Processing Units) started to appear as energy efficient compute accelerators. Nowadays GPUs are used in numerous fields of application other than graphics, such as electronic design automation, medical imaging, and signal processing. Programmable GPUs can not only be found in desktop computers, but also in mobile devices such as tablets and in supercomputers, having in common the need for a large amount of energy efficient compute power.

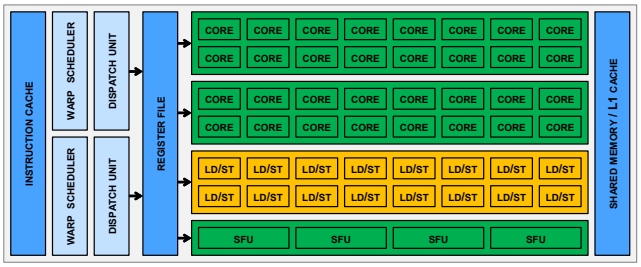


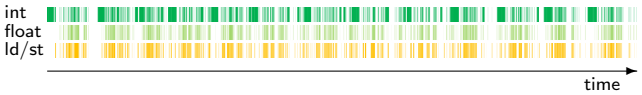
Figure 1: Streaming multiprocessor (SM) design of an NVIDIA Fermi GPU.

GPUs spend most of their hardware on many small (but heavily pipelined) 'cores', with no branch prediction, no speculative execution and only small caches. Instructions are issued in SIMD-style vectors, and latency is hidden by concurrently executing many independent vectors, resulting in a high-performance energy efficient SIMT architecture.

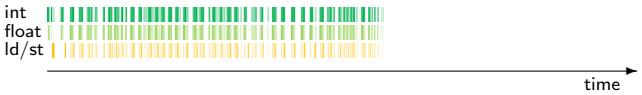
The number of cores on GPGPUs have increased from just over a hundred in 2006 [5] to thousands in 2013 [9], an increase of 21× in just 6.5 years. In the same period performance (GFLOPS) has increased 'only' 9×, and energy efficiency (GFLOPS/W) by a mere 5×. Also power consumption (TDP) has reached a ceiling of 250W since 2008, and at the same time clock frequency diminishes. This together reveals a trend in which more parallelism by more cores is preferred over clock frequency, i.e. more hardware is spent in order to increase performance and energy efficiency.

Simply adding more cores to a GPU does not result in an equivalent increase in performance or energy efficiency. Moreover, GPUs spend many cycles on data movement and control. In this work we propose an extension to the current GPU architecture in which the cores in an SM can be configured in a network with direct communication, creating a spatial computing architecture. Furthermore, each core executes a fixed instruction, reducing instruction fetch and decode count significantly. Data movement and control of an application is made implicit in the network, freeing up the cores for computations on actual data. By better utilizing the available cores, this results in increased performance and energy efficiency, while it only adds a relative small amount of hardware and preserves the original GPU functionality.

SCOPES '13, June 19-21, 2013, St. Goar, Germany



(a) Naive 3×3 convolution kernel



(b) Optimized 3×3 convolution kernel

Figure 2: SM activity over time for (a) a naive and (b) an optimized kernel. The activity is split in integer, floating point and load-store operations.

Section 2 presents a brief overview of a modern day GPU architecture. The proposed GPU-CC architecture is introduced in Section 3. In Section 4 we show the benefits of this architecture over the normal GPU architecture in a 2D convolution application. The paper ends with related work in Section 5 and conclusions and future work in Section 6.

2. BACKGROUND AND MOTIVATION

NVIDIA's Fermi GPU architecture [8] consists of multiple independent streaming multiprocessors (SM), sharing an off-chip memory. Each SM has a private instruction and data cache, a scratchpad (shared) memory, 32 cores, 16 load-store units, 4 special function units and two schedulers, see Fig. 1.

GPUs are programmed in an explicitly data-parallel language such as CUDA or OpenCL. The programmer writes code for a single thread, specifies how many threads have to be invoked and groups these threads in blocks, as only threads within a block can synchronize and share data via the shared memory.

As an example, consider the activity graph in Fig. 2 of an SM executing a 2D convolution kernel (see also Section 4). The SM's activity is split in three groups: (1) integer instructions representing address calculations and control operations, (2) floating point instructions on actual data and (3) load and store operations. Both the naive version (Fig. 2a) and the optimized version (Fig. 2b) start with address calculations, after which load instructions are issued. After an idle period the data arrives from the off-chip memory and floating point instructions are issued. The optimized kernel shows fewer load operations (and corresponding address calculations) than the naive implementation, due to the caching of data elements in registers (see Section 4.1).

Although the kernel in Fig. 2b is optimized and minimizes the number of memory loads, there are still idle cycles where the SM is stalled waiting for data, despite of the many threads it is executing to hide latency. Furthermore, a lot of cycles are spent on address calculations and load instructions rather than calculations on actual data. In 64% of the clock cycles at least one of the two schedulers in the SM is idle. Of the executed instructions 34% is used for floating point instructions on actual data, resulting in only 12% of the possible executed instructions over the duration of the kernel being spent on computations on actual data.

3. GPU-CC ARCHITECTURE

To better utilize the available cores in the GPU, we propose the GPU-CC architecture, which allows the cores in an SM to be configured in a network with direct communication, creating a spatial computing architecture. By moving data directly from one core to the next, data movement and control is made implicit in the network and instruction count can be reduced. Furthermore, each core is assigned one fixed instruction which it will execute during the whole kernel execution time. It is stored in a local configuration register and has to be loaded only once.

The standard GPU architecture is preserved, and no hardware blocks are removed. Hereby backwards compatibility for current GPU programs is assured, and programs which do not benefit of the GPU-CC architecture can use the standard GPU architecture as is. Only configuration registers and a communication network with FIFO buffers is added. The programmer can switch between the GPU's standard and GPU-CC architecture at run-time and specifies each core's GPU-CC instruction and connections in assembly by hand. We plan compiler support for future work.

The cores in an SM in the GPU-CC architecture are connected to each other via a communication network with FIFO buffers, as shown in Fig. 3. Via five data lanes, named **A** to **E**, cores can send data to each other's FIFOs. By passing data directly, the register file is not required and can be switched off. The multiplexers in the network are controlled by the configuration registers, creating a static circuit switched network for the duration of a kernel's execution.

In GPU-CC the register file and instruction fetch and decode units are switched off. According to the Integrated Power and Performance model of Hong and Kim [3] 12% of the power consumption of a GPU comes from these parts. Presumably more power is saved because cores execute a fixed instruction in GPU-CC, and not a mix of (integer and floating point) instructions. The power used by the communication network is expected to be low compared to the register file's power consumption, as it is smaller in memory size (see below) and consists of simple FIFO buffers instead of a multi-bank memory system with operand collectors. In GPU-CC not all cores are used in every application, which means some cores can be disabled saving more power.

Each core has three input FIFOs, as a core can execute instructions with (up to) three input operands. The load-store units have two input FIFOs, one for the address and one for the data in case of a store. All FIFOs have a size of 16 elements, only the address FIFO for the load-store units is 256 elements. These sizes are empirically determined, in future work we plan a more detailed evaluation.

Cores are triggered to execute an instruction when all input FIFOs have a data element available and when all FIFOs of the receiving cores have space available. The latency of a load operation in a load-store unit can be very long in case of a cache miss. The load-store unit only removes an item from its FIFO if the operation has completed. Therefore the input FIFO for the addresses is made (much) larger. The load-store unit has been equipped with a new prefetch element, which scans the address FIFO. When it detects an address with a new cache line address, it generates a memory request to fill the L1 cache with the corresponding cache line. This way the load-store units' following load operations will hit in the L1 cache, resulting in minimal stall cycles.

The main hardware costs of the GPU-CC architecture are the configuration registers and FIFO buffers. Each of the 32 cores has a configuration register and three 16 element FIFOs. Each of the load-store units also has an instruction cache, one 256 element and one 16 element FIFO.

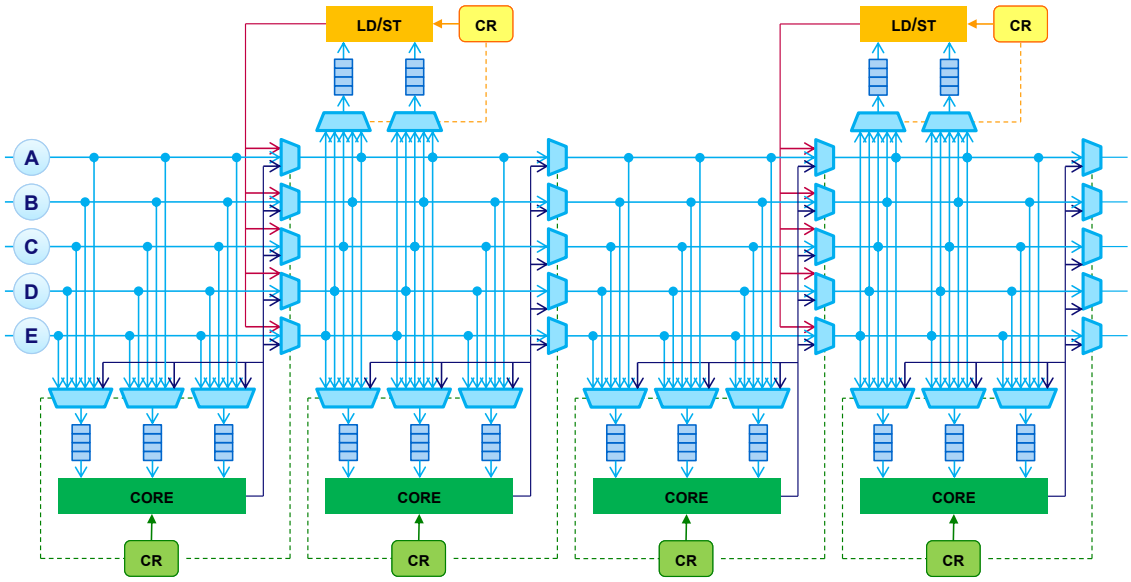


Figure 3: Design of the GPU-CC architecture. Cores and load-store units communicate via FIFO buffers and five data lanes named A to E. The single instruction each core executes is stored in a local configuration register (CR). Only four of the 32 cores and two of the 16 load-store units in an SM are shown for clarity.

This brings the total costs of the GPU-CC architecture to 195 kbit of storage per SM. An SM contains a register file of 32768 32-bit entries and a 64 kB scratchpad memory / L1 cache, making this 195 kbit an increase in storage of 12.4%. We note that multiple small FIFOs are more expensive in area than one large memory.

4. USE CASE: 2D CONVOLUTION FILTER

Convolution is a common operation in image and signal processing, among others. For example an image can be blurred by a 2D convolution with a Gaussian kernel. In this section we use a Gaussian filter to blur a 512×512 image. A mathematical representation is given in Eq. 1, where I is the input image and K the convolution kernel.

$$(I * K)(x, y) = \sum_i \sum_j I(x + i, y + j)K(i, j) \qquad (1)$$

First we show how a Gaussian filter is implemented on a conventional GPU. Next we implement the same filter on our proposed GPU-CC architecture. Finally we show the performance improvements of the proposed architecture.

4.1 2D convolution on a GPU

Execution results of five different versions of the 2D convolution kernel on an NVIDIA GTX 470 are shown in Table 1.

The *Naive* implementation reads nine input pixels and writes one output pixel from/to off-chip memory. Threads are organized in blocks of 16 by 16. This implementation only exploits the possible locality of the input pixels within the 16×16 block. In the other implementations, threads are organized in a vector 512 long, matching the width of the image. Since there are 14 SMs in the GPU used, each thread block processes a chunk of 36 or 37 (512/14) lines in the image, such that previously loaded lines can be re-used. In the *By line* implementations this re-use is achieved by relying on the L1 cache in each SM. In the *Shared memory* implementations the re-use is manually managed by loading rows of the image in the shared memory in the SM. The third and fifth implementation (annotated with *(R)*) use an extra level of re-use by keeping previously loaded lines in registers. All of these implementations, except *Naive*, outperform the NVIDIA CUDA SDK implementations of 2D convolution.

4.2 2D convolution using GPU-CC

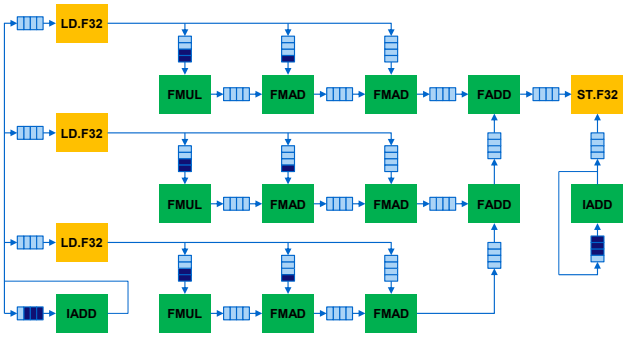
The GPU-CC architecture as introduced in Section 3 is implemented in GPGPU-Sim [1] version 3.1.2. The function each core executes is shown in Fig. 4a. The 3×3 structure of the convolution implemented here is visible in this figure. The FMUL, FMAD and FADD cores perform the multiply and add operations of the convolution. Two IADD cores are used for calculating the input and output addresses. Three LD.F32 cores are used to load the input data from the off-chip memory via the L1 cache. The instructions of these cores contain an immediate offset such that each load-store unit reads a different line in the image. The ST.F32 is used to store the output data. Fig. 4b shows how the cores can be placed in the communication network. Four out of the five data lanes are sufficient for the 2D convolution kernel.

Only 13 out of the 32 cores and 4 out of the 16 load-store units in each SM are used in the configuration of Fig. 4. This makes it possible to instantiate two copies of this configuration in each SM in order to improving performance.

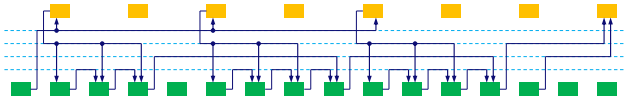
4.3 Experimental results

The GPU implementation (*By line (R)*) of the 3×3 convolution has a performance of 7.8 Gpixels/s. GPU-CC achieves a speed-up of 1.9× with a performance of 14.8 Gpixels/s. Considering each input and output pixel has to be transfered at least once, GPU-CC reaches 89% of the peak off-chip memory bandwidth. For a 5×5 convolution kernel the GPU-CC architecture attains a speed-up of 2.4× compared to an optimized implementation on a conventional GPU.

In the standard GPU implementation a total number of 220 thousand instructions are fetched, decoded and issued to



(a) functional implementation



(b) communication layout

Figure 4: Functional implementation (a) and communication layout (b) for a 3×3 2D convolution kernel in GPU-CC.

process the image of 512×512 pixels. The GPU-CC architecture uses 13 cores and 4 load-store units in the configuration of Fig. 4. Two copies of this configuration are instantiated on each SM, of which there are 14 on an NVIDIA GTX 470, making the total fetched instruction count as low as 476.

5. RELATED WORK

Reconfigurable architectures have been described in literature long before the introduction of GPGPUs. One example is the MorphoSys architecture [10], which consists of a main processor (RISC) and a reconfigurable processor array connected together via a bus. Another example is the ADRES architecture [6] which combines a main processor (VLIW) with a matrix of reconfigurable cells. The main processor and the reconfigurable array are separate hardware parts in the MorphoSys architecture. In the ADRES architecture several functional units of the reconfigurable matrix are shared with the VLIW processor, which reduces communication costs. As a result the ADRES architecture has two functional views, either the VLIW processor or the reconfigurable matrix is executing instructions. In our proposed architecture all resources are shared between the standard GPU mode and the proposed GPU-CC mode, keeping the original GPU functionality intact which is also used to setup the GPU-CC mode.

Two-level warp scheduling [7] reduces stall cycles due to long latency operations, just as GPU-CC's prefetch element in the load-store unit. Two-level warp scheduling issues instructions from a limited number of threads, just enough to hide the pipeline latency, until a long latency operation (e.g. off-chip memory load) is encountered, after which the instructions from other threads are executed.

6. CONCLUSIONS & FUTURE WORK

In this paper we proposed the GPU-CC architecture, adding an extra mode of computation to contemporary GPU architectures to better utilize its computational resources. By configuring the cores of a GPU in a network with direct

Table 1: Performance of five versions of 2D convolution (3×3) for a 512×512 image on an NVIDIA GTX 470 and on the GPU-CC architecture.

Version	Performance	Speed-up
Naive	3.5 Gpixels/s	1.0
By line	6.4 Gpixels/s	1.8
By line (R)	7.8 Gpixels/s	2.2
Shared memory	4.7 Gpixels/s	1.3
Shared memory (R)	4.7 Gpixels/s	1.3
GPU-CC	14.8 Gpixels/s	4.2

communication, performance is improved (1.9× and 2.4× for the 3×3 and 5×5 convolution example) while instruction fetch and decode count is reduced significantly, resulting in a reduced power consumption of an estimated 12%, at the cost of an extra 12.4% of memory space on the GPU.

In future work we plan a more thorough analysis of the FIFO buffer sizes, the number of data lanes and possibly other interconnect topologies for the GPU-CC architecture. We also plan to quantify the energy consumption benefits using GPGPU-Sim's power model [4]. Furthermore, we plan to improve the programmability and experiment with a range of applications, including applications which require more instructions than the number of cores available.

7. REFERENCES

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[2] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44(1):31–38, 2011.

[3] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *ISCA-37*, 2010.

[4] J. Leng, S. Gilani, T. Hetherington, A. ElTantawy, N. S. Kim, T. M. Aamodt, and V. J. Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *To appear in proc. of ISCA-40*, 2013.

[5] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2), 2008.

[6] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Field Programmable Logic and Application*, volume 2778 of *LNCS*. 2003.

[7] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *Micro-44*, 2011.

[8] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[9] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012.

[10] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, 49(5), 2000.