R-GPU: A Reconfigurable GPU Architecture

GERT-JAN VAN DEN BRAAK and HENK CORPORAAL, Eindhoven University of Technology

12

Over the last decade, Graphics Processing Unit (GPU) architectures have evolved from a fixed-function graphics pipeline to a programmable, energy-efficient compute accelerator for massively parallel applications. The compute power arises from the GPU's Single Instruction/Multiple Threads architecture: concurrently running many threads and executing them as Single Instruction/Multiple Data-style vectors. However, compute power is still lost due to cycles spent on data movement and control instructions instead of data computations. Even more cycles are lost on pipeline stalls resulting from long latency (memory) operations.

To improve not only performance but also energy efficiency, we introduce R-GPU: a reconfigurable GPU architecture with communicating cores. R-GPU is an addition to a 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 R-GPU data movement and control is implicit in the configuration of the network. Each core executes a fixed instruction, reducing instruction decode count and increasing energy efficiency. On a number of benchmarks we show an average performance improvement of $2.1\times$ over the same GPU without modifications. We further make a conservative power estimation of R-GPU which shows that power consumption can be reduced by 6%, leading to an energy consumption reduction of 55%, while area only increases by a mere 4%.

Categories and Subject Descriptors: C.1.4 [Processor Architectures]: Parallel Architectures; C.1.3 [Processor Architectures]: Other Architecture Styles—*Data-flow architectures*

General Terms: Design, Performance

Additional Key Words and Phrases: GPGPU, reconfigurable architecture

ACM Reference Format:

Gert-Jan van den Braak and Henk Corporaal. 2016. R-GPU: A reconfigurable GPU architecture. *ACM Trans. Archit. Code Optim.* 13, 1, Article 12 (March 2016), 24 pages.
DOI: <http://dx.doi.org/10.1145/2890506>

1. INTRODUCTION

Single-core performance growth halted in 2004 [Fuller and Millett 2011] with processors reaching their power consumption limit. Multi-core processors replaced the single-core ones and Graphics Processing Units (GPUs) started to appear as energy-efficient

Extension of Conference Paper “GPU-CC: a Reconfigurable GPU Architecture with Communicating Cores” [Van den Braak and Corporaal 2013]. The main contributions added in this article are:

- quantification and evaluation of the R-GPU architecture parameters, including the sizes of various FIFOs and the number of communication channels;
- a hardware evaluation of chip area, power consumption and energy efficiency;
- a tool which makes programming the R-GPU architecture much easier;
- an improved evaluation of the R-GPU architecture, including more benchmarks.

Authors' addresses: G.-J. van den Braak, Electronic Systems Group, Department of Electrical Engineering, Eindhoven University of Technology, P.O. Box 513, Flux 4.078, 5600MB Eindhoven, The Netherlands; email: g.j.w.v.d.braak@tue.nl; H. Corporaal, Electronic Systems Group, Department of Electrical Engineering, Eindhoven University of Technology, P.O. Box 513, Flux 4.134, 5600MB Eindhoven, The Netherlands; email: h.corporaal@tue.nl.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

2016 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 1544-3566/2016/03-ART12 \$15.00

DOI: <http://dx.doi.org/10.1145/2890506>

Fig. 1. Core count, GFLOPS and maximum power of various high-end GPUs at their time of introduction and their corresponding trend lines since 2006.

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 [NVIDIA Corporation 2013] such as mobile phones, tablets, and supercomputers. They all have in common the need for a large amount of energy-efficient compute power.

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

The number of cores on General Purpose GPUs (GPGPUs) have increased from just over a hundred in 2006 [Lindholm et al. 2008] to thousands in 2013 [NVIDIA Corporation 2012b], an increase of $21\times$ in just 6.5 years. In the same period performance (GFLOPS) has increased “only” $9\times$, and energy efficiency (GFLOPS/W) by a mere $5\times$. Power consumption has reached a ceiling of 250W since 2008, and at the same time clock frequency has diminished. This together reveals a trend in which more parallelism by more cores is preferred over clock frequency, that is, more hardware is spent in order to increase performance and energy efficiency. This trend is clear from Figure 1, where the number of cores, compute performance, and power consumption of a number of GPUs introduced since 2006 is shown.

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 R-GPU, an extension to the current GPU architecture in which the cores in a streaming multiprocessor 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 in a GPU, this results in increased performance and improved energy efficiency, while it only adds a relatively small amount of hardware. Since the original GPU functionality is preserved, R-GPU can still run existing GPU programs.

Fig. 2. (a) Streaming multiprocessor (SM) design of an NVIDIA Fermi GPU, consisting of two warp schedulers and two dispatch units, a register file, two groups of cores (CORE), one group of load-store units (LD/ST), one group of special function units (SFU), and a scratchpad (shared) memory. (b) A GPU consisting of 5 SMs and three memory partitions (GDDR) with L2 cache, all connected by a network on chip.

The main contributions of this work are as follows:

- We design, implement, and quantify the R-GPU architecture;
- We show an average performance improvement for benchmarks which benefit from the R-GPU architecture of $2.1\times$ over a regular GPU architecture;
- Tools to assist the programmer in programming the R-GPU architecture;
- An estimation of the hardware costs in terms of area and power consumption, including a tradeoff between hardware resources and performance, resulting in an energy consumption reduction of 55%.

The remainder of this work is organized as follows: Section 2 presents an overview of modern-day GPU architecture. The proposed R-GPU architecture is introduced in Section 3, as well as its programming model. The main benefits of the R-GPU architecture are described in Section 4. Section 5 introduces the tools developed to program the proposed architecture, and Section 6 contains a performance, area, and power evaluation. Related work is discussed in Section 7 and a summary is given in Section 8.

2. BACKGROUND

NVIDIA's Fermi GPU architecture [NVIDIA Corporation 2009] consists of multiple independent streaming multiprocessors (SM). Each multiprocessor consists of two groups of 16 cores, one group of 16 load-store units and a group of 4 special function units (SFU). It also contains a scratchpad memory (called shared memory by NVIDIA) and a level 1 data cache. Instructions are issued to the groups of cores, load-store units and SFUs by two warp schedulers and dispatch units, which share an instruction cache. A schematic overview of an SM is given in Figure 2(a). All SMs are connected to a shared L2 cache and off-chip memory (GDDR) via a network on chip (NoC), as shown in Figure 2(b).

GPUs are programmed in an explicitly data-parallel language such as CUDA or OpenCL. The programmer writes code, called a kernel, for a single thread (or work-item in OpenCL terminology), specifies how many threads have to be invoked, and groups these threads in equally sized thread blocks (or work-groups in OpenCL terminology). Only threads within a thread block can synchronize and share data via the scratchpad memory.

Each thread block is executed on a single SM in vectors of 32 threads, called a warp in CUDA terminology. Every two clock cycles both warp schedulers issue an instruction from an eligible warp to one group of cores, load-store units, or SFUs. One scheduler issues instructions from the even-numbered warps, and the other from the

Fig. 3. SM activity for (a) a baseline and (b) an optimized kernel. The activity is split in integer (int ■), floating point (float ■), and load-store (ld/st ■) operations.

odd-numbered warps. The GPU attempts to hide pipeline- and memory latency by scheduling instructions from other warps while the current instruction is in progress.

For a GPU to achieve peak compute performance, both schedulers have to issue an instruction every two cycles. Sometimes a scheduler cannot issue an instruction, because hardware (e.g., the single group of load-store units) is used by the other scheduler or because input operands are not yet available, due to either pipeline or memory latency.

As an example, consider the activity graph in Figure 3 of a multiprocessor of an NVIDIA GTX480 (Fermi architecture) executing a two-dimensional (2D) convolution kernel. The SM's activity is split into three groups: (1) integer instructions representing address calculations and control operations (e.g., calculating loop indexes and branches), (2) floating point instructions representing calculations on actual data, and (3) load and store operations. Both the baseline version (Figure 3(a)) and the optimized version (Figure 3(b)) 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 (data computations) are issued. The optimized kernel shows fewer load operations (and corresponding address calculations) than the baseline implementation, due to the caching of data elements in registers. The optimized version finishes earlier but despite all optimizations the GPU is still idle for a large part of the time.

Although the kernel in Figure 3(b) is optimized and minimizes the number of memory loads, the SM is still stalled waiting for data for 49% of the execution time, despite the many threads it is executing to hide latency. The two schedulers in the SM only utilize 37% of the possible instruction issue slots to execute instructions. Furthermore, many cycles are spent on address calculations and load instructions, and only 33% of the executed instructions are floating point instructions on actual data. This results in only 12% of the possible issued instructions over the duration of the kernel being spent on computations on actual data.

3. R-GPU ARCHITECTURE

The goals of the R-GPU architecture are twofold: first, the relative number of executed instructions spent on actual data computations is improved and, second, the number of stall cycles due to long latency memory operations is reduced. Applications with a regular access pattern and/or some form of reuse of their input data show a large gain in performance from the R-GPU architecture. For example, a 3×3 convolution application uses every input 9 times in a fixed pattern, as discussed in Section 6.1.1. Applications which have a limited level of parallelism due to data dependencies imposed by the algorithm benefit most from the R-GPU architecture. An example is the integral image application described in Section 6.1.3. Applications that are compute bound, such as

Fig. 4. Design of the R-GPU architecture. Cores and load-store units communicate via FIFO buffers and six data lanes named **A** to **F**. 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.

matrix-matrix multiplication, will not benefit from the R-GPU architecture, as R-GPU does not add any compute capabilities to the GPU.

To better utilize the available cores in the GPU, the R-GPU architecture configures the cores in an SM in a network with direct communication between them, 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 static 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. Just like regular GPU instructions each instruction can be predicated using a predicate register.

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 R-GPU architecture can use the standard GPU architecture as is. Only configuration registers (CR) and a communication network with FIFO buffers is added, see Figure 4. The GPU can switch between the standard and the R-GPU architecture at runtime. In a kernel which uses the R-GPU architecture, the GPU starts in its standard mode. After all configuration registers are filled and FIFO buffers are initialized, the GPU switches to the R-GPU mode. When it completes, it can switch back to regular mode if required.

The cores in an SM in the R-GPU architecture are connected to each other via a communication network with FIFO buffers, as shown in Figure 4. Via six data lanes, named **A** to **F**, cores can send data to each other's FIFOs. Each data lane is a unidirectional ring and is split in slots using muxes. Compute cores read input values from one slot in the data lanes and write to the next slot, as illustrated by the blue arrows out of each CORE in Figure 4. Load store units on the other hand read from one slot, but write to the same slot, as illustrated by the red arrows out of each LD/ST. This connection scheme makes it possible to calculate addresses in core N , load values in load-store unit $N + 1$, and process the loaded values in core $N + 1$. By passing data directly between cores and load-store units, 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.

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. The sizes of the FIFO are determined in Section 6.4.

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. In some cases a core can write its results back to its own FIFO, for example, when an increment instruction is mapped to a core. To completely hide the latency of a core, the FIFO size should be at least as large as the latency of the core. According to GPGPU-Sim [Bakhoda et al. 2009] the latency of a two- and three-input instruction is 8 and 10 cycles respectively (for integer or single precision floating point operands).

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 the head of its FIFO if the operation has completed. The common data type on a GPU is 32 bit, and an L1 cache line is 128 bytes wide [NVIDIA Corporation 2012a]. This means that up to 32 consecutive addresses (for 32bit words) can fall into the same cache line. 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. For the prefetcher to be able to prefetch a cache line, the address FIFO needs to hold at least 32 addresses or even more to be able to prefetch more cache lines.

The prefetcher requests an address within a cache line, which causes the cache line to be fetched from memory and placed in the cache. When the actual memory access occurs, the prefetch action may not have completed. In this case latency is reduced, and the final latency observed is somewhere between the cache latency and the main memory latency. In the unlikely event that a cache line is evicted between a prefetch request and an actual access, the memory access will take the full latency of a memory access. This usually does not happen, as there are few load-store units per SM (16 in Fermi) and 16kB (or 48kB) of cache per SM (depending on the cache configuration), which means there are at least 8 (or 24) cache lines (of 128 bytes) available for each load-store unit. This number will improve if fewer load-store units are used for load-instructions, assuming a perfect cache-placement policy. In the experiments the cache-placement policy of GPGPU-Sim is used, which mimics the actual GPU cache behavior of a set-associative cache.

The hardware added by R-GPU will change the hardware design of an SM. It will be larger because of the added hardware, and the maximum possible clock frequency of the GPU could be reduced. The exact timing impact on the SM is hard to predict, as the design of the NVIDIA Fermi SM on which R-GPU is based is not publicly available.¹ The performance of R-GPU is insensitive to latency. In addition, the data lanes can be split into segments to reduce the length of a single wire and minimize the timing impact. Also, as already observed in Section 1, the clock frequency of GPUs has diminished over the last couple of years to stay within the power budget. Furthermore, many GPU card manufacturers offer overclocking tools to users which allows them to increase the GPU's clock frequency at the costs of a higher power consumption. These two observations combined makes us believe that there is slack available in the GPU's clock period. A small increment in the required timing for the SM design caused by

¹An open source GPU called MIAOW is available from the Vertical Research Group at the University of Wisconsin—Madison: <http://miaowgpu.org/>. It implements a compute unit and is based off of the publicly released Southern Islands ISA by AMD.

Fig. 5. Three options for inter SM communication.

the added hardware of R-GPU therefore has no effect on the GPU's clock frequency. However, we can only be sure when this is verified with an actual GPU hardware design. However, this verification is outside the scope of this article.

The additional hardware parts in R-GPU consume extra power next to the GPUs regular hardware. But when a GPU runs in R-GPU-mode, the register file and instruction fetch and decode units can be switched off. This alone saves more power than the R-GPU hardware costs, as is elaborated in Section 6.5. Presumably more power is saved because cores execute a single, static instruction in R-GPU and not a mix of instructions. Furthermore, not all cores are used in every application in R-GPU, which can be disabled, saving even more power.

3.1. Inter-SM Communication

The communication network as described in the previous section only allows cores within one SM to communicate with each other. In case an R-GPU kernel consists of more instructions than there are cores available in a single SM, multiple SMs can work together to execute the kernel. Three options are investigated to make inter-SM communication possible.

The first option is direct communication between SMs; for example, each SM could have a connection to its direct neighbors, as shown in Figure 5(a). Although this could lead to a low-latency, high-throughput connection between SMs, it also requires thread blocks to be mapped to specific SMs, which is the exact opposite of the SIMT programming model (e.g., CUDA and OpenCL). In current GPU manufacturing, GPUs are made with the maximum number of SMs the architecture supports, but when some SMs fail during testing the GPU is sold as a lower-end model with the faulty SMs disabled. This would make the direct SM communication very hard to correctly implement in every possible situation.

The second option, shown in Figure 5(b), is communication via the network on chip (NoC). This option can handle different numbers of SMs in a GPU. Moreover, the thread blocks do not have to be mapped to specific SMs, as long as the communication via the NoC can be addressed using thread block IDs, instead of SM IDs. The downside of this options is that the NoC becomes much larger. Currently the NoC only supports memory requests (loads and stores) from the 16 SMs to the 6 memory partitions. To allow for communication between the SMs, the number of destinations in the NoC increases from 6 to 16+6.

The last and most applicable solution is communication via the L2 cache as shown in Figure 5(c). Like the second solution any number of SMs can be present in the GPU, and thread blocks do not have to be mapped to specific SMs. The sending SM can write to the global, off-chip memory which will be cached in the L2 cache. The receiving SM can read from the same memory and will get the data from the L2 cache. Load and

store instructions in a GPU have various cache operators which are used to specify in which level(s) of cache to update a certain request. For example, loads can be executed without touching the cache in case a data element is used only once, or stores can be marked as write-through to make them available for others in the off-chip memory as soon as possible. For the receiving SM, we introduce a new cache operator called *wait-for-hit*, which will remain in the L2 access queue until a hit occurs. The sending SM can use the already available cache operators. Loads and stores not involved in inter-SM communication can use cache operators to bypass the cache to prevent cache pollution which could influence the communication.

The inter-SM communication of Figure 5(c) requires memory loads to be held in the L2 cache access queue until a write to the same memory location occurs. Each L2 partition in an NVIDIA Fermi GPU (six in total in a GTX480) has a queue of eight entries. The *wait-for-hit* load instructions keeps circling through this L2 cache access queue and the L2 cache until the write from the sending SM has occurred. Deadlocks are prevented by having fewer pending load operations with the wait-for-hit cache operator than there are entries available in the queue. As a load-store unit that issues these *wait-for-hit* load instructions can only issue the next instruction after the previous one has finished, there is a very limited number of instructions which will be pending in the L2 caches and access queues. The benchmarks (INTEGR and NW) in Section 6 use only one *wait-for-hit* load instruction per SM.

3.2. Programming Model

An R-GPU program consists of two parts. The first part is a regular CUDA or OpenCL program which is executed in the GPU's standard mode. In this standard mode all configuration registers can be loaded and the FIFO buffers can be filled with initial values if required. Then the GPU can switch to R-GPU mode, which is done at a barrier instruction (e.g., `__syncthreads()` in CUDA). At a barrier instruction all threads in a thread block are at the same point in the kernel. To make sure all threads executing on an SM are synchronized at this point, only one thread block is allowed to execute at an SM at the same time. Note that if more thread blocks are used than there are SMs available, then multiple thread blocks can be executed on the SM after each other. Thread blocks can run in any order in regular GPGPU programs, as thread blocks are independent. To prevent deadlocks in kernels that use inter-SM communication, the R-GPU architecture requires thread blocks to be executed in a known order. The simulator used (GPGPU-Sim, see Section 5.3) orders thread blocks by increasing thread block-id.

Now the GPU starts executing in R-GPU mode in which the concept of threads is no longer used. The cores and load-store units in each SM execute the instruction stored in their configuration register for a given number of iterations, let's say N . For example, a load-store unit will issue N times a load operation with N addresses (which can all differ or be equal, depending on the application). The core receiving data from this load-store unit will calculate N values, which are then passed on to the next core.

Cores with no dependencies on other cores, and with initial values in their FIFOs, start executing first. Usually these are address calculating cores. The generated addresses arrive at the address FIFOs of the load-store units, which start fetching data from memory. After the data arrives, it is written to the FIFOs of cores who require these values. These cores start calculating, and forward the generated results to the next cores. Usually the results end at a load-store unit which is configured with a store-instruction. This core will store the calculated result back to the memory. Alternatively the final result can also be kept in a FIFO, which happens in reduction applications, such as summing all values in a matrix. After all cores and load-store units have executed their instruction N times, the GPU may switch back to its normal execution

Fig. 6. Block diagram of a three-tap FIR filter (a) and an R-GPU implementation (b). Two cores (IADD) are calculating addresses, three cores (FMUL and FMADD) are performing calculations on data, and two load-store units (LD and ST) are used.

```

void fir(float *input, float *coefficients, float *output) {
    for(int i=2; i<LENGTH; i++) {
        output[i] = input[i] * coefficients[0]
                  + input[i-1] * coefficients[1]
                  + input[i-2] * coefficients[2];
    }
}

```

Listing 1. Sequential C-code for a 3-tap FIR filter.

mode. Now the results which are kept in the FIFOs can be used again in the regular GPU mode.

4. R-GPU MOTIVATION

The R-GPU architecture improves GPU performance in two ways: First, it removes redundant memory loads by having the cores communicate directly with each other using FIFO buffers. Second, it improves the obtained memory bandwidth for applications with a low level of parallelism. These two benefits are discussed in more detail in this section.

4.1. Benefit 1: Removing Redundant Memory Loads

In a three-tap FIR filter as shown in Figure 6(a), three input values are combined into one output value. A sequential C-implementation is given in Listing 1. In a simple GPU implementation, one thread would be launched for a single output element. This implies that each thread has to load three input values. Thread N loads input values N , $N-1$, and $N-2$, thread $N+1$ loads input values $N+1$, N , and $N-1$, and so on. Even though not all these loads result in off-chip memory accesses due to the caches present in a GPU, all threads still have to issue the three load instructions to acquire all input values. This results in all input values being loaded 3 times. In an ideal situation each input value is loaded only once. To limit the number of load instructions, each thread can calculate multiple output elements. Previously loaded values can be kept in registers and can be reused for multiple output elements. This approach still implies that (some) input values are copied from one register to the other, depending on the amount of loop unrolling.

Another approach in reducing the total number of loads is to allow threads to use each others input values. NVIDIA's Kepler architecture introduces "shuffle" instructions [NVIDIA Corporation 2012b] which allow threads in a warp to read each others registers [Nickolls et al. 2010]. This has a limited effect on reducing the number of loads, as only the 32 threads in a warp can communicate. Boundary conditions have to

be taken into account; the first and the last thread in a warp have to read extra values as they do not have neighbors from which they can read.

In R-GPU, no redundant loads and no register copies are required in the FIR filter example. Each input value is read only once and is directly forwarded to the cores which need the data. Figure 6(b) shows the R-GPU implementation of the same three-tap FIR filter. Data elements are loaded via the LD load-store unit and stored in the FIFO buffers of the three taps of the FIR filter, implemented by the FMUL and FMAD cores. To ensure that the correct input values N , $N - 1$, and $N - 2$ are used for output value N , the input FIFOs of these three cores have 0, 1, and 2 initial values, respectively. The final results are stored using the ST load-store unit. Addresses for the load-store units are generated by the two IADD cores.

As the number of load-store units in a GPU's streaming multiprocessor (SM) is relatively small,² reducing the number of load and store instructions has a large impact on execution time. Especially in applications in which the memory bandwidth is the main bottleneck, the load-store units are used most (if not all) of the time. For these applications the compute instructions can be scheduled in the memory latency period of the load and store instructions. Reducing the number of load and store operations for these applications will result in a performance improvement. Regular optimization techniques (e.g., calculating multiple output values per GPU thread) already show some improvement. R-GPU can further reduce the number of load and store instructions, resulting in a larger performance improvement.

4.2. Benefit 2: Improving Memory Bandwidth

In a regular GPU, each thread's execution (e.g., calculating a FIR filter) starts with loading data from memory into registers. When all data have arrived, the registers are used in calculating the output values of the FIR filter. These are then subsequently stored back in the memory. Only after the store operations have been issued are the registers free and can new values be loaded from memory in the registers. This causes a delay in the processing, as every thread is waiting for input data to arrive. GPUs attempt to hide this waiting time by running many threads in parallel. Often this is not enough to hide all memory latency. Measurements on a GTX470 show that at least 768 threads per SM are required to achieve more than 90% of the obtainable bandwidth (which is achieved by running 1536 threads, the maximum number of threads per SM) for a simple, memory-bound kernel. As described above, it is possible to reuse input elements, and to apply loop-unrolling to improve performance, but still threads may stall waiting for data to arrive from memory. This effect is clearly visible in applications which have a low level of parallelism, and cannot run many threads in parallel, as illustrated in the Needleman-Wunsch and integral image benchmarks described in Section 6.

Two-level warp scheduling [Narasiman et al. 2011] is one way to reduce idle cycles due to the long latency (load) operations. It schedules instructions only from a limited number of threads, just enough to hide the pipeline latency, until a long latency operation is encountered. Only then are the instructions from other threads executed, which fill the idle cycles caused by the load operation as much as possible.

In R-GPU, on the other hand, the address calculating IADD cores (e.g., in the FIR filter of Figure 6(b)) never have to wait until registers are free. As long as there is space available in the FIFOs connected to these cores' outputs, more addresses can be generated. This ensures that the load-store unit will load data as quickly as possible,

²An NVIDIA Fermi SM has 16 load-store units compared to 32 compute cores. An NVIDIA Kepler SM has only 32 load-store units compared to 192 compute cores.

Fig. 7. Integral image application programmed in the visual programming environment (a) and the resulting mapping to cores and load-store units (b).

until the FIFOs connected to its output are full. Only when R-GPU starts executing a kernel a short stall period is observed in which the first data elements are loaded.

As the off-chip memory latency on a GPU is hundreds of cycles, a prefetch element is added in R-GPU to each load-store unit. It scans the address FIFO and creates memory requests when an address of a new cache line is found. These requests fill the L1 cache in the SM, and subsequent loads from the load-store unit will hit in the cache.

5. PROGRAMMING TOOLS

For ease of programming R-GPU, programming tools have been developed to help the programmer. A visual programming environment is developed as a front end (Section 5.1), and a back end is described in Section 5.2. This back end automates the error prone task of mapping instructions to cores and assigning the correct data lanes between cores. It can also make a tradeoff between the number of cores and the number of data lanes used. A full compiler will be part of future work. Finally, the performance of R-GPU is evaluated using a simulator described in Section 5.3.

5.1. Front End

A front end visual programming environment is developed in which the programmer can draw instructions as boxes onto a canvas. Dependencies between instructions are drawn as arrows between the boxes. Initialization values for the instructions' operands also can be given. To help the programmer even more, pseudoinstructions are added. For example, an increment instruction is included, which will map to an ADD instruction. The programmer can specify the step size, the number of initial values, and the start-offset, which can be a function of the thread block id and a kernel parameter.

In Figure 7(a), an R-GPU implementation of the integral image application (see also Section 6.1.3) is drawn in the visual programming environment. The corresponding sequential C-code is given in Listing 2. Each thread block calculates one row in this example. Two increment units are used to generate the addresses to load and store

```

void integral_image(char *in, int *out) {
    for(int y=1; y<HEIGHT; y++)
        for(int x=1; x<WIDTH; x++)
            out[y*WIDTH+x] = out[ y *WIDTH + x-1]
                            + out[(y-1)*WIDTH + x ]
                            - out[(y-1)*WIDTH + x-1]
                            + in [ y *WIDTH + x ];
}

```

Listing 2. Integral image sequential C-code.

from. The first one is used to address 32bit integers, and the second one addresses 8bit integers. An offset to a kernel parameter is given, which is a pointer to an array in the off-chip memory. Also an offset for each specific thread block is specified to ensure that each thread block processes a different part of the data. Both increment units have five initial values in their input FIFOs. Only two additions (ADD) and one subtraction (SUB) are required to calculate each output, similar to Listing 2. The elements loaded from the row $(y - 1)$ above the current row being processed are used twice, once for (x) and once for $(x - 1)$. The second input has one initial value to account for the difference in the index. The result of one calculation is used as the input for the next, this is done via the loop back arrow of the last ADD unit, shown in the bottom right of Figure 7(a).

5.2. Back End

The back end maps instructions to cores and load-store units and assigns the data lanes in two steps. It can make a tradeoff between the number of cores and the number of data lanes used. The results are written to a file to be used by the simulator. A visual overview of the mapping is given, as shown in Figure 7(b).

In the first step, a mapping of instructions to cores and load-store units is made by constraint programming. The mapping has to satisfy the number of cores and load-store units in an SM, as well as the number of data lanes. Also data dependencies are taken into account, as the communication network is a directed ring between the cores.

Each instruction has to be mapped to a distinct core or load-store unit. A constraint is added between two instructions if one requires the output of the other; the sending instruction has to be placed on a core which is connected to the core with the receiving instruction. Due to the unidirectional data lanes, the sending core has to be to the left in Figure 4 of the receiving core. Special care has to be taken with load-store instructions. Where cores read at one data lane slot and write to the next, load-store units write at the same slot as they read from (see also Figure 4).

The instruction mapping is constrained by the number of data lanes available. To limit the number of data lanes used, a second set of constraints is added. For each communication between a writing and a reading core, a virtual data lane is set to be occupied between the writing and the reading core. For each slot in the data lanes, the total number of virtual data lanes used has to be less than or equal to the actual number of data lanes available.

In the second step, the cores which have to communicate their results to other cores are assigned to the actual data lanes. This is also done by constraint programming. As the total number of data lanes used is limited in the first step, a standard geometrical packing constraint can be used.

5.3. Simulator

The R-GPU architecture is based on NVIDIA's Fermi architecture, the latest GPU architecture supported by GPGPU-Sim [Bakhoda et al. 2009]. Version 3.2.1 of this

cycle level simulator has been modified to be able to simulate the R-GPU architecture. Switching from the standard GPU execution model to the R-GPU execution model can be done at a barrier instruction (e.g., `__syncthreads()` in CUDA). After a barrier instruction finishes all threads in a thread block are synchronized, making it the ideal point in time to switch the execution model of an SM. The number of cycles it takes to finish all threads is modeled in the simulator, just like the loading of the initial values in the FIFOs.

6. EVALUATION

To validate the performance improvements of R-GPU we implemented a number of kernels from a range of applications using the tools described in Section 5. All benchmarks are listed in Table I. From GPGPU-Sim 3.2.1 [Bakhoda et al. 2009], we use the neural network benchmark; from Rodinia 2.4 [Che et al. 2009], the Needleman-Wunsch, Pathfinder, and Streamcluster benchmarks; and from Parboil 2.5 [Stratton et al. 2012], the MRI-Q and stencil benchmark. We also added the 2D convolution from Van den Braak and Corporaal [2013], the histogram kernel from Gómez-Luna et al. [2013], and a newly implemented integral image benchmark. The sizes of the inputs used in the evaluation are also stated in Table I. The last five columns in this table list the number of data lanes used (labeled **A** to **F** in Figure 4), the number of cores and load-store units used in each implementation, how many times each implementation is replicated in each SM, and whether inter-SM communication (see Section 3.1) is used. Note that all benchmarks contain fewer instructions than the number of cores and load-store units in an SM. All benchmarks are compiled using NVIDIA's CUDA compiler `nvcc` version 4.2, the latest version supported by GPGPU-Sim. All benchmarks are tested using the modified version of GPGPU-Sim 3.2.1 (see Section 5.3) using the configuration file for an NVIDIA GTX480 GPU supplied with GPGPU-Sim.

Three benchmarks are described in Section 6.1. The performance of R-GPU is evaluated in Section 6.2. In Sections 6.3 and 6.4, the communication network is discussed and the sizes for the data and address FIFOs are determined. Finally, in Section 6.5 a conservative power and area estimation is given.

6.1. Benchmarks

All benchmarks are implemented using the tools described in Section 5. If a benchmark requires fewer cores than available in an SM, the implementation is replicated to fill up the SM as much as possible. The implementation of most benchmarks (e.g., 2D-convolution and stencil) utilizes the spatial locality available in these benchmarks. Other benchmarks have more complex reuse patterns, such as the MRI-Q benchmark. The inter-SM communication is demonstrated by the integral image and Needleman-Wunsch benchmarks. These three benchmarks are discussed below.

6.1.1. 2D-Convolution. 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. A mathematical representation is given in Equation (1), where I is the input image and K the convolution kernel,

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

The R-GPU implementation is shown in Figure 8(a). The 3×3 structure of the convolution implemented here is visible in this figure. The nine FMUL and FMAD cores perform the multiply and add operations in the convolution. Two extra FADD cores are used to sum the results together. 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

Table 1. List of Benchmarks Used to Evaluate and Quantify the R-GPU Architecture

Benchmark	Benchmark set	Description	Input size	Data lanes	# core instr.	# Id/st instr.	repl. per SM	inter SM comm.
2D convolution (CONV)	GPU-CC	2D 3×3 image blur	512×512	4	13	4	2	no
Histogram (HIST)	Gómez-Luna et al. 2013	Histogram of a Full HD gray scale image	1920×1080	2	6	3	4	no
Integral (INTEGR)	-	Prefix sum	2048×2048	3	5	3	4	yes
MRI-Q (MRIQ)	Parboil	Magnetic Resonance Imag.	$32 \times 32 \times 32$	4	14	4	2	no
Neural Network (NN)	GPGPU-Sim	Third layer of a 5-layer neural network	1250×128	3	13	11	1	no
Needleman-Wunsch (NW)	Rodinia	DNA sequence alignment	2048×2048	3	5	3	4	yes
PathFinder (PATH)	Rodinia	Grid Traversal	131072×128	6	19	6	1	no
Stencil (STENCIL)	Parboil	3D stencil operation	$128 \times 128 \times 32$	5	14	8	2	no
Streamcluster (STREAM)	Rodinia	Sum of squared distances calculation	16384 data points	6	19	13	1	no
								no

Fig. 8. Functional implementation (a) and communication layout (b) for a 3×3 2D convolution kernel in R-GPU. Cores with integer instruction are dark green, and those with floating point instruction in light green. Cores and load-store units which are not active are shown as an outline only in the communication layout.

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. Figure 8(b) shows how the cores can be placed in the communication network. Four of the six data lanes are sufficient for this 2D convolution kernel.

Only 13 of the 32 cores and 4 of the 16 load-store units in each SM are used in the configuration of Figure 8. This makes it possible to instantiate two copies of this configuration in each SM in order to improving performance.

6.1.2. MRI-Q. In the MRI-Q benchmark, a matrix Q is computed, representing the scanner configuration for calibration [Stratton et al. 2012]. The arrays Q_r and Q_i are calculated according to Equations (2) and (3). The arguments of the sin and cos functions are reused, similarly to the regular GPU implementation. For every output indexed by n , a single value of the arrays x , y , and z is used, as well as all values of the array in . In the R-GPU implementation eight values of x , y , and z are loaded at setup time in FIFOs, and at runtime each value of the in array is used eight times, once for each value of x , y , and z in the FIFOs. A new value of the in array is only read every eight cycles. Two of these implementations can be mapped in a single SM, resulting in the total number of thread blocks required to be $n/(8 \cdot 2)$. In this way, the in array gets reused as much as possible. Array in (3072 entries in the Parboil input dataset) fits in the L1 cache of the GPU, and therefore the in array is only read from the off-chip memory once.

$$Q_r[n] = \sum_k in[k].\phi \cdot \cos((in[k].x \cdot x[n] + in[k].y \cdot y[n] + in[k].z \cdot z[n]) \cdot 2\pi), \quad (2)$$

Fig. 9. Speed-up of R-GPU compared to an optimized GPU implementation.

$$Q_i[n] = \sum_k in[k].\phi \cdot \sin((in[k].x \cdot x[n] + in[k].y \cdot y[n] + in[k].z \cdot z[n]) \cdot 2\pi). \quad (3)$$

6.1.3. Integral Image. The integral image, or summed area table, of a matrix M contains the sum of all pixels above and to the left of the current element, as shown in Equation (4):

$$I(x, y) = \sum_{\substack{x' \leq x \\ y' \leq y}} M(x', y'). \quad (4)$$

In the R-GPU implementation, a single row of the output I is calculated by only eight cores, and therefore each thread block can work on four rows. The dependency between row n and row $n + 1$ in two different thread blocks is resolved using inter-SM communication. The first thread block will write its results to the memory via the L2 cache. The second thread block will stall until it can read this data. Although this leads to a long (functional) pipeline, this approach requires the input data to be read only once, and only one in four output rows have to be reread from the L2 cache. In a standard GPU implementation usually the integral image is calculated in two steps. First, the horizontal integral image is calculated, after which the intermediate output is written to memory. Second, the intermediate output is read again, and the vertical integral image is calculated.

6.2. R-GPU Performance

The benchmarks as listed in Table I are implemented on the R-GPU architecture using the tools of Section 5. Their performance is compared to a regular GPU as simulated in GPGPU-Sim [Bakhoda et al. 2009] which uses the greedy-then-oldest (GTO) warp scheduler. A reference implementation for the regular GPU implementation is taken from the benchmarks suites. When multiple reference implementations were available, all are optimized and tested and the best one is used as the reference. All reference implementations are highly optimized, not only thread and thread block sizes but also loop unrolling factor are tuned for GPGPU-Sim. For example, the performance of the Neural Network benchmark taken from GPGPU-Sim is improved more than $8\times$. For the 2D-convolution benchmark five different reference implementations are implemented and evaluated, with an execution time difference between them over $2\times$.

The speed-up of the R-GPU implementation over a regular GPU is shown in Figure 9. All these benchmarks benefit from the R-GPU architecture. Benchmarks that do not gain performance can use the regular GPU architecture as such, and does not (have to) experience a slow-down. The geometric mean of the speed-ups for the benchmarks shown in Figure 9 is $2.1\times$.

Fig. 10. Dependencies between matrix elements in the Needleman-Wunsch algorithm. Seven steps are required to calculate this 4×4 matrix.

Fig. 11. Activity of the cores within one SM in R-GPU over the execution of the 2D-convolution application. The activity is split in integer (int), floating point (float), and load-store (ld/st) operations.

Needleman-Wunsch (NW), a nonlinear global optimization method for DNA sequence alignments, shows the largest performance improvement of all benchmarks. It fills a 2D matrix from top left to bottom right as illustrated in Figure 10. At the start only the first row and the first column of the matrix are filled. To calculate one element in the matrix, the elements to the northwest-, north- and west-adjacent are required, similarly to the integral image benchmark of Section 6.1.3. At every step of the algorithm the next diagonal is calculated. This means that the available parallelism is limited, the maximum parallelism is reached when calculating the main diagonal of the matrix. This limited parallelism limits the performance of a regular GPU implementation greatly. The reference implementation from Rodinia uses a two-level approach. The matrix is split in tiles, and each tile is calculated as described above, and the tiles are processed in the same manner. Like the other benchmarks the reference implementation is optimized for GPGPU-Sim. The R-GPU architecture can transfer the calculated value of one matrix element directly to its adjacent elements. Similarly to the integral image implementation of Section 6.1.3 the R-GPU implementation of NW calculates four output rows per thread block. Dependencies between thread blocks are resolved via inter-SM communication (Section 3.1). R-GPU's fine-grained communication, combined with the removal of redundant reads and writes to the off-chip memory, lead to the large speed-up of $11.5\times$ over a regular GPU.

The integral image benchmark has a similar dependency pattern as the Needleman-Wunsch benchmark. However, the calculations of the integral image benchmark are linear. Therefore, the reference GPU implementation can be split into two kernels for the horizontal and vertical sums. These kernels contain more parallelism and far fewer redundant load and store operations than the reference implementation of the Needleman-Wunsch benchmark. Hence the speed-up for the integral image benchmark is lower but still $3.4\times$.

In Figure 11, the activity of the CONV benchmark is shown over the duration of the kernel's execution. The activity of the cores is split into INT and FLOAT instructions, representing computations on addresses and data respectively. Because the CONV benchmark

Fig. 12. R-GPU area costs of the communication network (4 to 10 lanes), address FIFOs (64 to 512 entries) and data FIFOs (4 to 32 entries).

is limited by off-chip memory bandwidth, not all cores are active all the time, as is clear from the gaps in Figure 11. Compared to the activity graph of the regular GPU in Figure 3 the R-GPU architecture manages to issue load instruction to the off-chip memory constantly, instead of in many short bursts. This is the main source of the speed-up of $1.35\times$ of R-GPU.

6.3. Communication Network

The R-GPU architecture in Figure 4 has six data lanes; the number of lanes required for each benchmark is shown in Table I. Most benchmarks require only two, three, or four data lanes, while STENCIL requires five. Only PATH and STREAM require all six data lanes. In these two cases there is a “hotspot” where all data lanes are used, most often fewer data lanes are used. For example, see the instruction mapping in Figure 7(b), where the maximum number of data lanes is only used for a short period. In case an application requires more data lanes than available in the architecture it is possible to rewrite the application such that data values are recomputed instead of communicated.

The wires in the data lanes consume a large portion of the area used by R-GPU, as elaborated in Section 6.5. The data lanes only consume power when used, since unused data lanes can be switched off. Therefore, the number of data lanes in the R-GPU architecture is an area-performance tradeoff, which is shown in Figure 12.

The current implementation of the PATH benchmark uses 6 data lanes, but alternative implementations use either 4 or 10 lanes. The 4-lane implementation is twice as slow, and the 10-lane implementation is only 14% faster. Increasing the number of data lanes from 6 to 10 increases the area cost of R-GPU from 9mm^2 to 15mm^2 , as shown in Figure 12. Not only the area used by the data lanes increases but also more muxes are added in the data lanes. Furthermore, the muxes connecting the communication network to the FIFOs increase in size as they require more inputs.

6.4. FIFO Sizes

A range of FIFO sizes were tested for all benchmarks to find the best possible tradeoff between performance and FIFO size, for example, number of entries, area, and power. The data FIFO’s size is tested with 4, 8, 16, and 32 entries as shown in Figure 13(a). The execution time of each benchmark is normalized against the execution time of a FIFO size of 8 entries. Most benchmark are not sensitive for the different FIFO sizes. The NN benchmark requires at least 5 entries in a data FIFO, and the MRIQ and PATH benchmarks take twice as long to execute when the FIFO contains only 4 entries. Therefore, we select the data FIFO size to be 8 entries. At this size the PATH, NW and NN

Fig. 13. Normalized execution time for various FIFO sizes. The data FIFO sizes range from 4 to 32 (a) and the address FIFO sizes range from 64 to 512 (b).

benchmarks perform 15%, 8%, and 6% slower than at a FIFO size of 16 respectively. All other benchmarks' performance is within 3% of its best value.

Increasing the number of entries in the data FIFOs from 8 to 16 entries only increases the area of R-GPU from 2.3mm^2 to 2.6mm^2 , as shown in Figure 12. But the power consumption of R-GPU increases by 8% due to the larger FIFOs, while performance hardly improves for most benchmarks.

The address FIFO size is tested with 64, 128, 256, and 512 entries, as shown in Figure 13(b). The execution time of each benchmark is normalized against the execution time of a FIFO size of 256 entries. Four benchmarks, HIST, MRIQ, PATH, and STREAM, perform significantly better with larger FIFO sizes, while CONV, NN, and STENCIL only show small performance improvements. The INTEGR and NW benchmarks perform the same for all FIFO sizes, as they are limited by the communication between multiprocessors (Section 3.1) and cannot use the prefetch capabilities of the load-store unit. The STREAM benchmark uses an array-of-structs as its inputs, which contain four 32bit words. In R-GPU one load-store unit is used for each word. Consecutive addresses for each word are 16 bytes apart, instead of the normal 4 bytes for 32bit words. This causes the load-store unit's prefetcher to prefetch more cache lines than usual, which causes significant cache pollution and a slow-down for a FIFO size of 512 entries for the STREAM benchmark. Taking all the above considerations into account, a FIFO size of 256 entries is chosen for the address FIFOs.

Increasing the number of entries in the address FIFOs from 256 to 512 entries increases the area of R-GPU from 7.6mm^2 to 9.8mm^2 , as shown in Figure 12. But the power consumption of R-GPU increases by 26% due to the larger FIFOs, while performance only improves for the HIST benchmark.

6.5. Power and Area Estimation

To estimate the power savings of R-GPU, we first estimate how much power can be saved by switching off some parts of the GPU, for example, the register file and the instruction cache and fetch- and decode-units. Second, we give a detailed estimation of the power the R-GPU architecture requires using Cacti, Verilog synthesis, and a wire power model. Finally, an area estimation of the R-GPU architecture is given.

6.5.1. Power Savings. According to GPUWattch [Leng et al. 2013], the register file takes 13.4% of the dynamic power in a GTX480 (average over multiple compute benchmarks), which is about 13W. In an older Quadro FX5600 the dynamic power consumed by the register file is 17.2%. Similar numbers are reported by GPUSimPow [Lucas et al. 2013], where the register file of a GTX240 consumes 12.6% of its power in the Black-Scholes benchmark from the CUDA SDK, while the instruction fetch- and decode-units take 5.65% of the GPU's power. The Hong and Kim power model [Hong and Kim 2010] estimates the power of the register file and the instruction fetch- and decode-units to

Fig. 14. Power distribution of the added hardware by R-GPU in two benchmarks: memory bound (a) and compute bound (b) and the area distribution of the added hardware by R-GPU (c).

be 7.9W or 4.5% and 13W or 7.5% respectively for a GTX280 averaged over a number of benchmarks.

Although the power numbers reported are for different GPUs, and even different GPU architectures, the combined power consumed by the register file and instruction fetch- and decode-units of the GPU range from 12% to 18%. Since we use the same GPU (GTX480) for our simulations as Leng et al. [2013], we use the 13W of dynamic power reported by Leng et al. [2013] as a conservative estimation of the power we can save by switching off the register file and instruction fetch- and decode-units.

6.5.2. Power Costs. To estimate the power consumption of the R-GPU architecture we modeled the muxes and FIFOs according to the configuration shown in Figure 4 with six data lanes. The muxes and FIFO logic are implemented in Verilog, and power numbers are obtained using the Cadence Encounter RTL Compiler v11.20 and a 40nm standard cell library. The SRAM memories in the FIFOs are modeled using CACTI 6.5 [Muralimanohar et al. 2007] using a 45nm technology size. The total number of data and address FIFOs in each SM is 112 (3 for each of the 32 cores and 1 for each of the 16 load-store units) and 16 respectively, with 8 and 256 entries each respectively (Section 6.4). As the wires in the interconnection network between the cores of an SM can become quite long, we model the power consumption of the wires separately. A normalized energy per bit of 60pJ/m is used, similarly to Gebhart et al. [2011] where also architecture changes in the same GPU (GTX480) are proposed. This is well within the range of 20 to 100pJ/m reported in Zhang et al. [2011] for transmission line types of interconnection structures at the 45nm technology node. The normalized energy per bit for a repeated RC wire type of interconnect is much higher, approximately 400pJ/m according to Zhang et al. [2011]. Similarly to Gebhart et al. [2011], we base the wire length on a die photo of a GTX480 and assume it to be 8mm, the length plus the width of an SM.

The final power consumption is dependent on the benchmark used, similarly to Hong and Kim [2010], Leng et al. [2013], and Lucas et al. [2013]. For the STENCIL and MRIQ benchmarks the power consumption for the different parts of R-GPU is shown in Figure 14. The STENCIL benchmark is limited by the off-chip memory bandwidth, and therefore the R-GPU pipeline often stalls. When the pipeline stalls, the wires and

values in the FIFOs do not toggle, leading to a moderate power consumption of 1.4W. The address FIFOs use a relative large amount of the power because all load-store units are used, but not all cores in the SMs in this benchmark. The MIRQ benchmark, on the other hand, is not limited by off-chip memory bandwidth and the R-GPU pipeline is stalled much less, leading to a power consumption of 3.2W. As 28 of the 32 cores are used, and only 8 of the 16 load-store units, the power consumption of the data FIFOs is relatively large for the MIRQ benchmark compared to the STENCIL benchmark.

Combining the 13W of dynamic power which can be saved by switching off the register file, the 3.2W of added power in the R-GPU architecture and the total (average) GPU power consumption of 153W [Leng et al. 2013] leads to a power saving of 6%. Note that this is a conservative estimation, as power saving due to disabling the instruction cache and fetch- and decode units are not taken into account. Moreover, the fact that each core is executing a static instruction for a period of time is not taken into account in this estimation.

6.5.3. Area Costs. The area costs of the R-GPU architecture are estimated similarly to the power costs in the previous section. The area values as reported by CACTI 6.5 are used to determine the area of the FIFOs. The logic area is estimated using Verilog simulations using the Cadence Encounter RTL Compiler and the area of the long wires in the data lanes is estimated using the wire model in Zhang et al. [2011]. The total area costs of the R-GPU architecture is estimated to be 23mm², which is an increase of just 4% compared to the total area of 529mm² of an NVIDIA GTX480.

An overview of the area costs for the different components of R-GPU is shown in Figure 14(c). The wires in the data lanes add the most area. This is mainly due to the estimation of the wire length we use and the sum of the length and width of an SM. When the actual length of these wires is smaller, the (relative) area costs would also be less as wire area scales linearly with the wire length [Zhang et al. 2011].

Although there are many more data FIFOs than address FIFOs in the R-GPU architecture (112 vs. 16 per SM), the address FIFOs take the most area. A single data FIFO contains eight 32bit values (Section 6.4) and has an area of only 0.0015mm². A single address FIFO contains 256 entries but also has two read ports instead of one to support prefetching (Section 3), resulting in an area of 0.031mm² per FIFO. Despite having two read ports the address FIFO requires less area per bit than the data FIFO.

7. RELATED WORK

Reconfigurable architectures have been described in literature long before the introduction of GPGPUs. One example is the MorphoSys architecture [Singh et al. 2000], which consists of a main processor (RISC) and a reconfigurable processor array connected together via a bus. Another example is the ADRES architecture [Mei et al. 2003] 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 R-GPU mode, keeping the original GPU functionality intact which is also used to setup the R-GPU mode.

Register file caching is introduced in Gebhart et al. [2011] as an alternative method to reduce register file energy. This is combined with a two-level thread scheduler which maintains a small set of active threads to hide ALU and scratchpad memory access latency. A larger set of pending threads is used to hide main memory latency. The

two-level scheduler combined with a six-entry per-thread register file cache reduces register file energy consumption by 36%.

Two-level warp scheduling [Narasiman et al. 2011] reduces stall cycles due to long latency operations, just as R-GPU'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. Combined with the large warp microarchitecture the two-level warp scheduling improves performance by 19%.

Specialization in software has been introduced by Bauer et al. [2011]. In this work certain warps in a thread block are used as a DMA to copy data from the off-chip DRAM memory to the on-chip scratchpad memory. The resulting speed-up for several benchmarks are $1.15\times$ – $3.2\times$. In "Singe" by Bauer et al. [2014], all warps in a thread block are assigned specific subcomputations of a kernel. This allows "Singe" to deal efficiently with the irregularity in both data access patterns and computation. It also makes large working sets in the on-chip scratchpad memory possible. The final performance result is a $3.75\times$ speed-up over their previous GPU implementations.

FCUDA [Papakonstantinou et al. 2009] adapts the CUDA programming model into an FPGA design flow, which maps the coarse- and fine-grained parallelism exposed in CUDA onto the reconfigurable fabric. CUDA kernels are compiled into an FPGA design using high-level synthesis tools. Where FCUDA targets an FPGA, and synthesizes its processing elements specific for the kernels, R-GPU uses a GPU as platform and re-uses the existing processing elements.

The Single-Graph Multiple Flows (SGMF) architecture presented in Voitsechov and Etsion [2014] looks similar to the R-GPU architecture. It is a complete redesign of a GPU as a coarse-grained reconfigurable fabric (CGRF), where R-GPU is an extension to the existing architecture, which can still be used by applications which do not map well to the R-GPU architecture. The functional units in SGMF are interconnected in a two-dimensional grid, compared to the one-dimensional, unidirectional data lanes in R-GPU. In SGMF the functional units also execute a static instruction but for different threads, like the standard GPU architecture. In comparison, the R-GPU architecture executes on thread block granularity and has no notion of threads. In SGMF values are tagged with thread IDs which allows threads to overtake each other. R-GPU does not use tagging, and values have to be processed in order. Similarly to R-GPU, the SGMF architecture does not require a central register file nor an instruction fetch- and decode-unit. The reported performance of SGMF is comparable to existing GPUs, while consuming 57% less energy on average. This is comparable to R-GPU, which achieves an energy consumption reduction of 55%. The difference is that R-GPU improves performance over $2\times$. In terms of area SGMF is much more efficient, with a reported size of 318mm^2 using a 40nm technology node for a configuration with 15 SGMF cores. For comparison, a reference GPU such as the NVIDIA GTX480 has an area of 529mm^2 , and R-GPU adds another 4% on top of this.

8. CONCLUSIONS

In this work R-GPU is presented, a new and reconfigurable GPU architecture with communicating cores. It is fully backwards compatible with existing GPUs. A communication network with FIFO buffers is added between the cores of an SM, which allows cores to directly send data to each other. Hereby data movement and control operations (e.g., loop calculations) are avoided. This leads not only to an improved performance for various benchmarks but also to an increased energy efficiency. The parameters of the architecture, such as the FIFO sizes, have been quantified using benchmarks from Rodinia and Parboil. Based on these benchmarks an average speed-up of $2.1\times$ is

measured over the regular GPU architecture. The added hardware of R-GPU costs only 4% of extra area. This new hardware also consumes power, but more power is saved as the register file and instruction fetch- and decode-units can be switched off. This leads to a conservative approximation of the power savings of R-GPU of 6%. Combined with the performance improvement this leads to an energy consumption reduction of 55%. To program the R-GPU architecture tools are presented, consisting of a visual programming environment and an instruction mapper based on constraint programming.

The current R-GPU architecture is based on NVIDIA's Fermi architecture. In future work we plan to expand this to more recent architectures, such as Kepler and Maxwell, in which "shuffle" instructions [NVIDIA Corporation 2012b] are introduced. This allows threads in a warp to read each others registers [Nickolls et al. 2010]. The available hardware for the "shuffle" instructions could be reused for the R-GPU architecture, reducing the area costs of R-GPU. Also a full R-GPU compiler is targeted for future work.

REFERENCES

- Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2009 (ISPASS 2009)*. IEEE, Washington, DC, 163–174. DOI : <http://dx.doi.org/10.1109/ISPASS.2009.4919648>
- Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. ACM, New York, NY, Article 12, 11 pages. DOI : <http://dx.doi.org/10.1145/2063384.2063400>
- Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: Leveraging warp specialization for high performance on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. ACM, New York, NY, 119–130. DOI : <http://dx.doi.org/10.1145/2555243.2555258>
- Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC'09)*. IEEE Computer Society, Washington, DC, 44–54. DOI : <http://dx.doi.org/10.1109/IISWC.2009.5306797>
- Samuel H. Fuller and Lynette I. Millett. 2011. Computing performance: Game over or next level? *Computer* 44, 1 (Jan 2011), 31–38. DOI : <http://dx.doi.org/10.1109/MC.2011.15>
- Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 235–246. DOI : <http://dx.doi.org/10.1145/2000064.2000093>
- Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. 2013. An optimized approach to histogram computation on GPU. *Machine Vision and Applications*. 24, 5 (2013), 899–908. DOI : <http://dx.doi.org/10.1007/s00138-012-0443-3>
- Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 280–289. DOI : <http://dx.doi.org/10.1145/1815961.1815998>
- Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 487–498. DOI : <http://dx.doi.org/10.1145/2485922.2485964>
- E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (March 2008), 39–55. DOI : <http://dx.doi.org/10.1109/MM.2008.31>
- Jan Lucas, Sohan Lal, Michael Andersch, Mauricio Alvarez-Mesa, and Ben Juurlink. 2013. How a single chip causes massive power bills GPU-SimPow: A GPGPU power simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 97–106. DOI : <http://dx.doi.org/10.1109/ISPASS.2013.6557150>
- Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo Man, and Rudy Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Field*

- Programmable Logic and Application*, Peter Cheung and George A. Constantinides (Eds.). Lecture Notes in Computer Science, Vol. 2778. Springer, Berlin, 61–70. DOI: http://dx.doi.org/10.1007/978-3-540-45234-8_7
- Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC, 3–14. DOI: <http://dx.doi.org/10.1109/MICRO.2007.30>
- Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, 308–317. DOI: <http://dx.doi.org/10.1145/2155620.2155656>
- J. R. Nickolls, B. W. Coon, M. Y. Siu, S. F. Oberman, and S. Liu. 2010. Single interconnect providing read and write access to a memory shared by concurrent threads. (16 2010). US Patent 7,680,988.
- NVIDIA Corporation. 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. (2009). http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.
- NVIDIA Corporation. 2012a. CUDA C Programming Guide 5.0. (Oct 2012). <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>.
- NVIDIA Corporation. 2012b. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. (2012). <http://www.nvidia.com/content/pdf/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.
- NVIDIA Corporation. 2013. NVIDIA Tegra K1: A New Era in Mobile Computing. http://www.nvidia.com/content/pdf/tegra_white_papers/tegra_k1_whitepaper_v1.0.pdf.
- A Papakonstantinou, K. Gururaj, J. A. Stratton, Deming Chen, J. Cong, and W.-M. W. Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Proceedings of the IEEE 7th Symposium on Application Specific Processors, 2009. SASP'09*. IEEE Computer Society, Washington, DC, 35–42. DOI: <http://dx.doi.org/10.1109/SASP.2009.5226333>
- H. Singh, Ming-Hau Lee, Guangming Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. 2000. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*. 49, 5 (May 2000), 465–481. DOI: <http://dx.doi.org/10.1109/12.859540>
- John A. Stratton, Christopher Rodrigues, I.-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial throughput Computing*. Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign, Urbana.
- Gert-Jan van den Braak and Henk Corporaal. 2013. GPU-CC: A reconfigurable GPU architecture with communicating cores. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems (M-SCOPES'13)*. ACM, New York, NY, 86–89. DOI: <http://dx.doi.org/10.1145/2463596.2486153>
- Dani Voitsechov and Yoav Etsion. 2014. Single-graph multiple flows: Energy efficient design alternative for GPGPUs. In *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*. ACM, New York, NY, 12.
- Yulei Zhang, Xiang Hu, Alina Deutsch, A. Ege Engin, James F. Buckwalter, and Chung-Kuan Cheng. 2011. Prediction and comparison of high-performance on-chip global interconnection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19, 7 (July 2011), 1154–1166. DOI: <http://dx.doi.org/10.1109/TVLSI.2010.2047415>

Received May 2015; revised December 2015; accepted February 2016