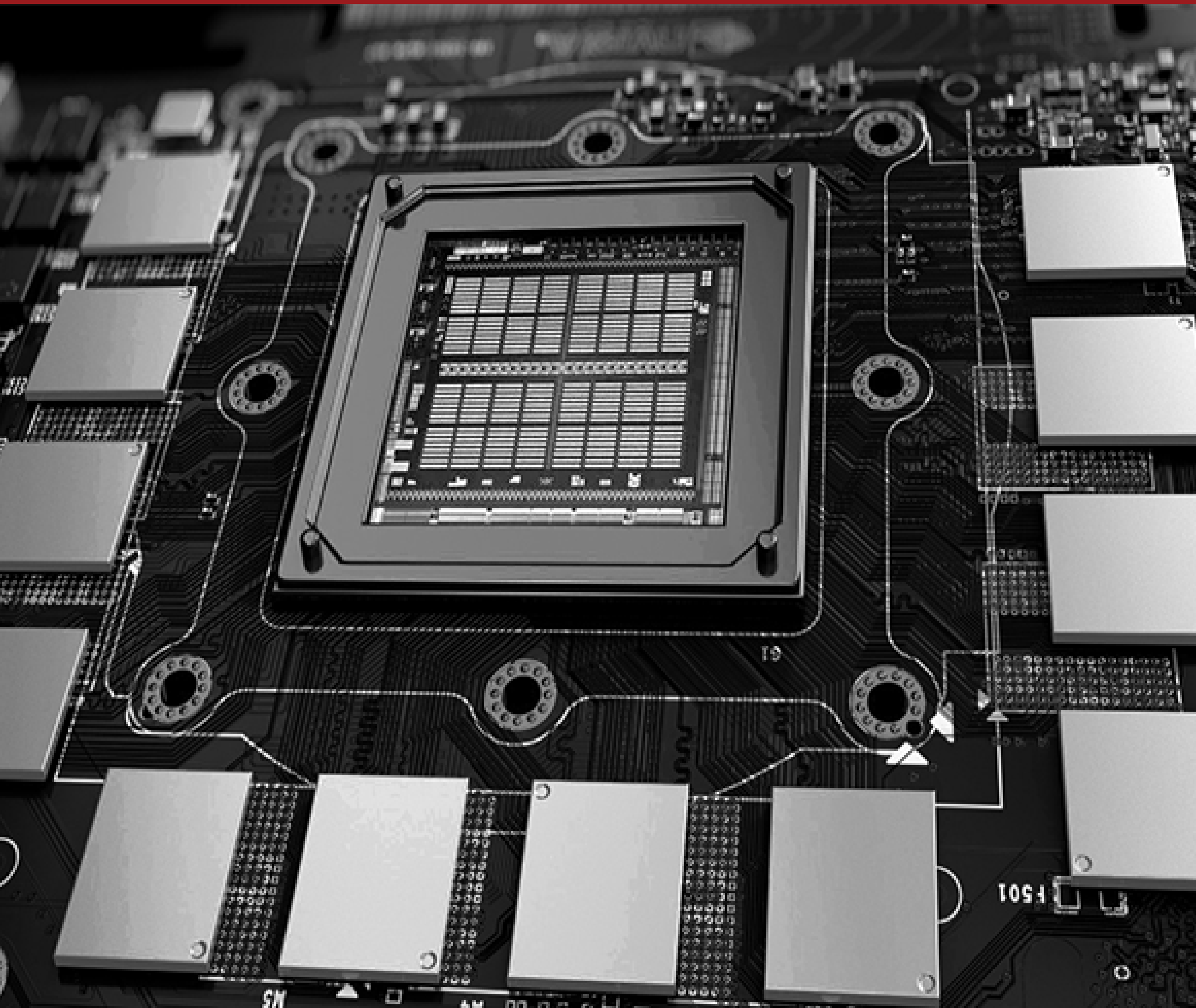
VIVIDSPARKS

PROGRAMMING GUIDE FOR VIVIDSPARKS FAMILY OF MANYCORES



Executive Summary

The GPU has become an integral part of today's mainstream computing systems. The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaces its CPU counterpart. The GPU's rapid increase in both programmability and capability has spawned a research community and industries that has successfully mapped a broad range of computationally demanding, complex problems to the GPU. This effort in general-purpose computing on the GPU, also known as GPU computing, has positioned the GPU as a compelling alternative to traditional microprocessors in high-performance computer systems of the future. Leaders in computing are now profiting from their investments in new number systems initiated half a decade ago. We introduce revolutionary GPU called RacEr based on POSIT™ number system. The POSIT number system is a tapered floating point with very efficient encodings for real numbers and two exceptional values, zero and infinity. The POSIT encoding leads to higher accuracy compared to floats at half bit-width, which leads to higher performance and lower cost for big-data applications. Furthermore, the POSIT standard defines rules for reproducibility in concurrent environments enabling high-productivity and lower-cost for software application development for multi-core and many-core deployments. In this document we give programming guidelines for VividSparks family of manycores. *The programming guidelines remains same for all products except number of cores in each product.* We have given programming guidelines using RacEr.

Introduction

RacEr GPU consists of an array of tiles, connected by a 2-D mesh network called the manycore accelerator, with an attached external memory and I/O system. Most tiles contain processing, memory, and communication routers. Processing in a tile is done with CPU cores and specialized accelerator cores The accelerator cores are added to personalize the RacEr architecture and to improve energy/performance for targeted applications. In addition to these tiles, the architecture features victim cache tiles, often located on the edge of the tiled array and termed column caches ($), but potentially located at other positions in the array. These victim cache tiles are in turn connected to memory controllers that interface to multiple parallel memory channels that go to DRAM -- high bandwidth memory (HBM), DDR4, or other. The Figure 1 on next page shows the high-level view. In some cases, some of the CPU cores might be replaced with accelerator tiles.

Each CPU core contains a 4KB direct-mapped instruction cache (1024 instructions), and a 4KB local data memory. The cores features non-blocking loads and stores, which allow them to overlap the memory latency to remote memories in the system while they execute non-dependent instructions. These word-level accesses go out onto the 2-D mesh network to the remote tile, cache or dram that owns the address in question.

Within the architecture, we have the concept of a tile group, which is a physically contiguous subarray of tiles. Tile groups work together to perform cooperative multiprocessing, where a group of cores shares a set of banked memories and distributes shared data structures across these banks. The cores use a bulk synchronous programming model, where a program is divided into phases in which each address is either read/write-owned by a single tile, read-owned by everybody, or requires atomic operation/mutex enabled atomic accesses. Between each phase, the tiles synchronize via a synchronization barrier. This allows a group of tiles to bring in a chunk of memory from DRAM, operate on that data in parallel with high locality, and then write it back to DRAM. Access to data structures within a tile group enjoys reduced energy usage, latency and increased throughput relative to access to data structures in DRAM, or in the column caches.

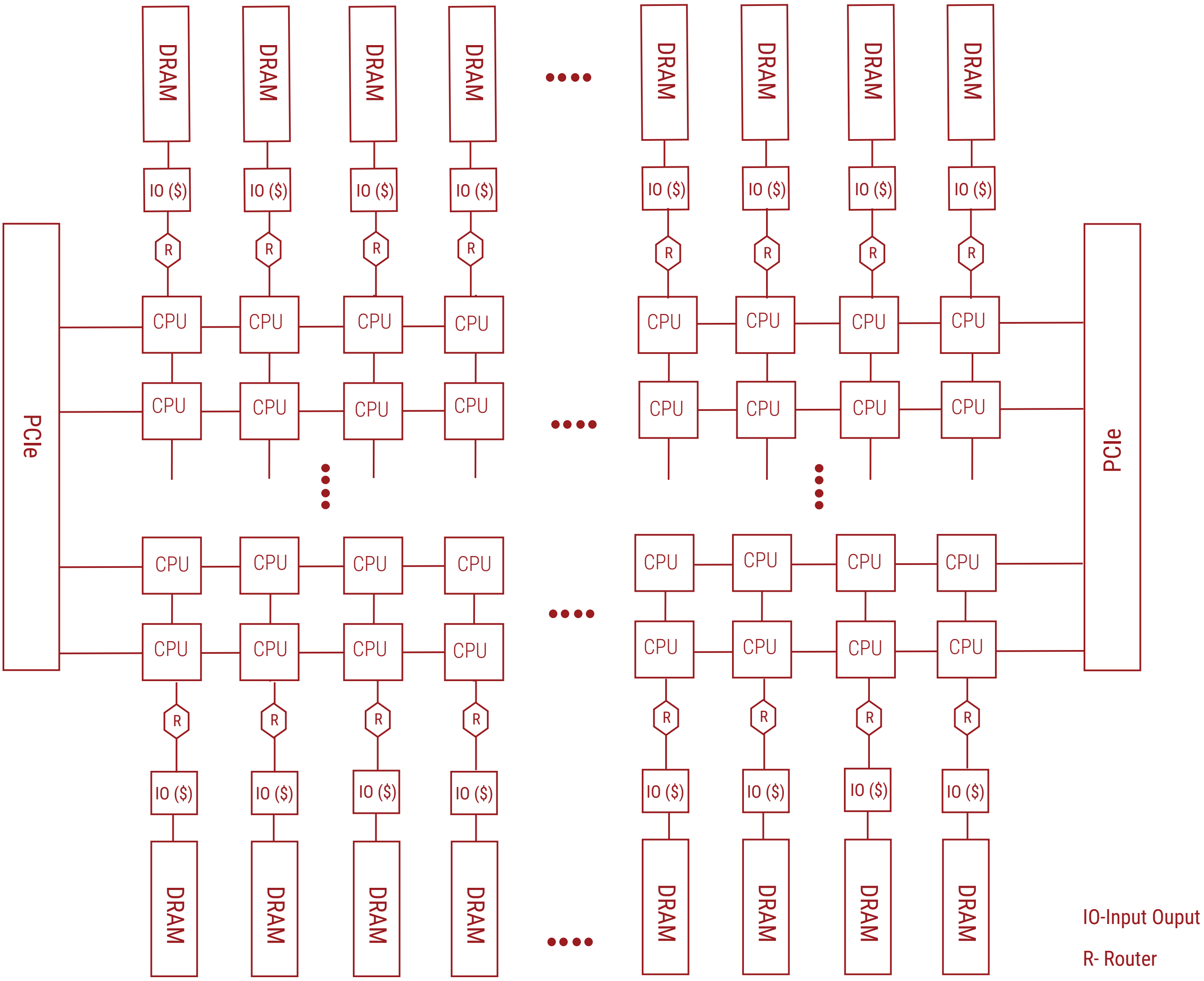


Figure 1: RacEr Architecture

RacEr API

The following functions in Tables 1 are defined for programmer convenience:

RacEr API

Macro	Definition
`RacEr_set_tile_x_y()`	Sets RacEr_X and RacEr_Y to the X and Y coordinate of the tile.
`RacEr_x_y_to_id(x,y)`	Calculates tile's flat id using its (x,y) coordinates.
`RacEr_id_to_x(x)`	Calculates tile's x coordinate using its flat id.
`RacEr_id_to_y(y)`	Calculates tile's y coordinate using its flat id.
`RacEr_remote_ptr (x,y,addr)`	Forms a remote address by taking in x and y coordinates of remote tile and the address to local variable. Used in RacEr_remote_store and RacEr_remote_load.
`RacEr_remote_store (x,y,addr,val)`	Stores val to the local address addr in the memory space of the tile at (x,y).
`RacEr_remote_store_uint8 (x,y,addr,val)`	Stores the 1-byte val to the local address addr in the memory space of the tile at (x,y).
`RacEr_remote_store_uint16(x,y, addr,val)`	Stores the 2-byte val to the local address addr in the memory space of the tile at (x,y).
`RacEr_remote_load (x,y,addr)`	Loads from the local address addr in the memory space of the tile at (x,y).
`RacEr_dram_ptr(addr)`	Forms a pointer to an element on the DRAM attached to the bottom of tile's column using the local address.
`RacEr_dram_load (addr,val)`	Loads from DRAM into val by using RacEr_dram_ptr.
`RacEr_dram_store (addr,val)`	Stores val into dram by using RacEr_dram_ptr.
`RacEr_tilegroup_ptr (addr,index)`	Takes in the local address of tilegroup-shared memory, and the array index. Calculates the coordinates of the tile holding that index, and returns a RacEr_remote_ptr to that element.
`RacEr_tilegroup_load (addr,index,val)`	Loads from tilegroup-shared memory into val by taking in its local address and index, using RacEr_tilegroup_ptr.
`RacEr_tilegroup_store (addr,index)`	Stores local val to tilegroup-shared memory by taking in its local address and index, using RacEr_tilegroup_ptr.
`RacEr_remote_control_store(x,y, addr,val)`	Remote stores the value into the instruction memory of tile (x,y) using local address.
`RacEr_remote_freeze(x,y)`	Starts the execution of tile (x,y) using RacEr_remote_control_store.
`RacEr_remote_unfreeze (x,y)`	Stops the execution of tile (x,y) using RacEr_remote_control_store.
`INIT_TILE_GROUP_BARRIER (ROW_BARRIER_NAME, COL_BARRIER_NAME, x_cord_start, x_cord_end, y_cord_start, y_cord_end)`	Initializes parameters for a barrier instruction for all tiles within tilegroup using the start and end coordinates of the tilegroup.
`RacEr_tile_group_barrier (ROW_BARRIER_NAME, COL_BARRIER_NAME)`	Synchronizes all tiles within the tilegroup by taking in the row and column barrier names generated by INIT_TILE_GROUP_BARRIER.
`RacEr_finish()`	Terminates simulation.
`RacEr_wait_while(cond)`	Wait for condition to be true.

Table 1

CUDA-Lite Programming Guide

Introduction

The cuda-lite programming extension allows RacEr to mimic the environment and structure of a GPU, and provides a parallel SPMD programming interface. A series of architectural and software upgrades are implemented to allow for a simple step-by-step transition from CUDA to RacEr code. In this section, we will compare the execution model of the two architectures, the hardware/software extensions to support CUDA-Lite in RacEr, then explain the necessary steps for CUDA to RacEr translation, and conclude with a series of examples.

Execution Model

CUDA follows the single-instruction-multiple-data (SIMD) programming model, in which threads are the basic units of execution. A kernel is launched with programmer-specified grid and block dimensions, in which threads are grouped into three-dimensional grid of three-dimensional blocks. Blocks are dispersed into GPU streaming multiprocessors (SMs) upon kernel launch. A block of threads assigned to an SM is then divided into fixed-length groups called warps, scheduled to be executed in a parallel lock-step manner, where each thread executes the same set of instructions on their own private data set. The scheduler unit in SM sequentially runs warps until all block threads finish their execution. Notice that the combination of parallel and sequential execution in this model is crucial, since running thousands of threads all at once will require unfeasibly large hardware resources.

The RacEr manycore forms a similar model with the same combination of parallel and serial thread execution observed in GPUs. To mimic the SPMD architecture of an SM in RacEr, CPU cores are grouped into tilegroups, a two-dimensional grid of adjacent tiles that run the same instructions on their private data memory. Each block of threads will be assigned to one tilegroup, with the threads evenly distributed among the cores. Parallel execution is achieved by running all cores in a tilegroup at the same time, while threads inside each core are executed sequentially, in a threadloop. Threadloops are in fact a loop that is wrapped around the kernel instructions, forcing the core to iterate over all threads assigned to it, and execute them sequentially. The Figure 2 below demonstrates the arrangement of threads into blocks and grids, and compares their execution strategy in both GPUs and RacEr. Execution model of GPU vs. RacEr. Threads in a Kernel are organized into a grid of blocks, which are dispatched to SMs in GPU and tilegroups in RacEr for execution.

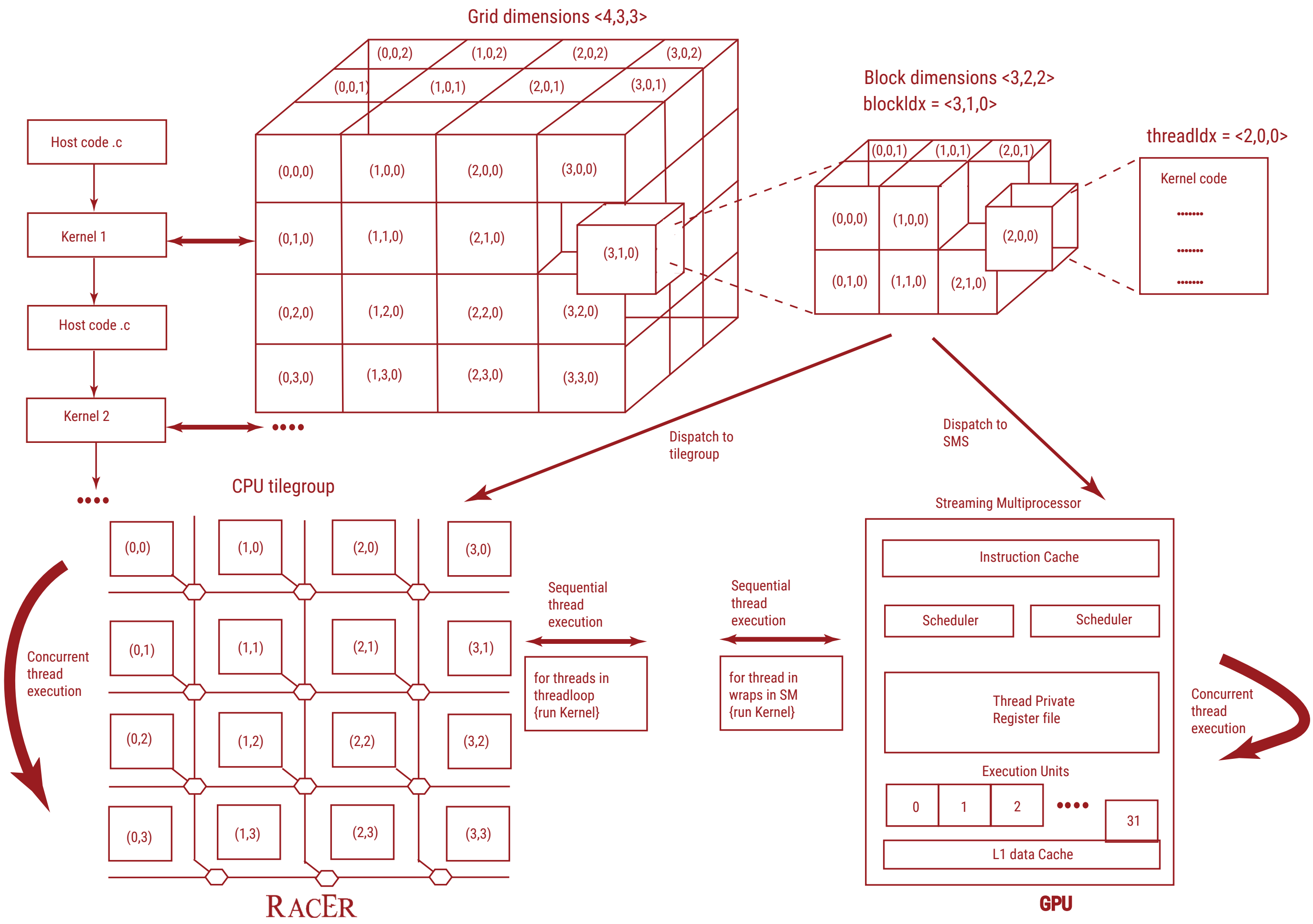


Figure 2: CUDA-Lite Architecture

Hardware/Software Extensions

Tilegroup-Shared Memory

In a GPU, threads in a block have access to

1. hread-private registers,
2. Block-shared memory,
3. Global memory through a hierarchy of caches.

RacEr implements similar memory hierarchy by use of

1. Private data memory in each core,
2. Tilegroup-shared memory
3. Global DRAM banks connected to the bottom of each column

Tilegroup-shared memory is referred to a section of each core's data memory, banded together and accessible from all cores by remote load/store operations over the manycore network. The close proximity of cores in a tilegroup ensures small delay cycles for load/store operations.

The Programmer can declare tilegroup-shared memory with language-specific macros in the RacEr programming interface. The hardware handles the distribution of shared space among the cores, and generates hash functions necessary to access the tilegroup-shared array from any core in tilegroup.

Threadloops

Cores in a tilegroup will be responsible for running multiple threads of a block. To make that possible, we follow an approach similar to MCUDA to transform a CUDA code to a sequential program covered by nested loops, with each iteration representing a thread. A series of steps is needed to achieve this feature.

Passing Known Variables

A series of CUDA known variables are defined, and their value is passed to RacEr code.

gridDim	Grid dimensions <x, y, z>
blockDim	Block dimensions <x, y, z>
blockIdx	Position of block in grid <x, y, z>

RacEr code also has access to its own hardware specific variables, color coded as Maroon.

RacEr_tilegroupDim	tilegroup dimensions <x, y, z=1>
RacEr_x, RacEr_y, RacEr_z	position of core in tilegroup (RacEr_z = 0)
RacEr_num_threads_per_core	<x, y, z> = blockDim / RacEr_tilegroupDim

Introducing Nested Threadloops

- The kernel code is wrapped in a series of nested loops, one for each dimension of block.

- Thread index variables (threadIdx.<x,y,z>) are replaced by loop iterators (iter_x, iter_y, iter_z) in kernel code.

- The number of threads per core depends on the blockDim / RacEr_tilegroupDim ratio for each dimension. Notice that the z dimension of tilegroup is always 1, and RacEr_z always resolves to 0. That means every core is responsible for running all corresponding z dimensions of block at all times.

Thread Synchronization - Loop Fission

- CUDA enforces synchronization among threads in a block by use of the __threadsynchronize() primitive, stalling further execution until all threads have reached a certain point in the code. RacEr achieves the same functionality by employing two levels of synchronization:

    - Synchronization among cores in a tilegroup: enforced by use of RacEr_tile_group_barrier primitive already implemented in RacEr hardware.

Original CUDA Kernel

```
__global__ void original_kernel(**argv)
{
        ...
        // body (threadIdx.z, threadIdx.y, threadIdx.x)
        ...
}
```

Kernel After Step1

```
__global__ void kernel_step1(**argv)
{
        // threadloops
        for (it_z = RacEr_z; it_z < blockDim.z;
              it_z += RacEr_tilegroupDim.z){
               for (it_y = RacEr_y; it_y < blockDim.y;
                     it_y += RacEr_tilegroupDim.y){
               for (it_x = RacEr_x; it_x < blockDim.x;
                     it_x += RacEr_tilegroupDim.x){
                        ...
                        // body (it_z, it_y, it_x)
                        ...
                }
        }
        }
}
```

- Synchronization among threads in a single core: implicitly held by the sequential execution model of a threadloop.

- Threadloop is an implicit barrier for all threads in core:

    - Each iteration executes a thread to its end.
    - Then suspends the thread until other threads have reached the same point.

- For a synchronization primitive inside a threadloop, RacEr breaks the already existing threadloop into two to enforce an implicit barrier, one for instructions before the sync, and one for those after it.

Thread Synchronization - Deep Fission

- What if sync instruction is inside a for loop, while loop, or if/else construct?
- The following algorithm (adopted from MCUDA) can resolve all synchronization conflicts in a CUDA code:

Kernel After Step 1

```
__global__ void kernel_step1(**argv)
{
      // threadloops
      for (it_z = RacEr_z; it_z < blockDim.z;
           it_z += RacEr_tilegroupDim.z){
            for (it_y = RacEr_y; it_y < blockDim.y;
                 it_y += RacEr_tilegroupDim.y){
                  for (it_x = RacEr_x; it_x < blockDim.x;
                       it_x += RacEr_tilegroupDim.x){
                        ...
                        // body1 (it_z, it_y, it_x)
                        ...
                        __syncthreads();
                        ...
                        // body2 (it_z, it_y, it_x)
                  }
            }
      }
}
```

Kernel After Step 2

```
__global__ void kernel_step1(**argv)
{
      // threadloop 1
      for (it_z = RacEr_z; it_z < blockDim.z;
           it_z += RacEr_tilegroupDim.z){
            for (it_y = RacEr_y; it_y < blockDim.y;
                 it_y += RacEr_tilegroupDim.y){
                  for (it_x = RacEr_x; it_x < blockDim.x;
                       it_x += RacEr_tilegroupDim.x){
                        ...
                        // body1 (it_z, it_y, it_x)
                        ...
                  }
            }
      }

-----------------Implicit Barrier ---------------------

      // threadloop 2

      for (it_z = RacEr_z; it_z < blockDim.z;
           it_z += RacEr__tilegroupDim.z){
            for (it_y = RacEr_y; it_y < blockDim.y;
                 it_y += RacEr__tilegroupDim.y){
                  for (it_x = bsg_x; it_x < blockDim.x;
                  it_x += RacEr__tilegroupDim.x){
                        ...
                        // body2 (it_z, it_y, it_x)
                        ...
                  }
            }
      }
}
```

Translate for loops into while loops, extract initialization and conditions. Initialize the fifo S: add all synchronization statements to the fifo S Perform on every element S_i in S:

```
loop:
    if immediate scope containing $S_i$ is not a threadloop
            partition scope into three sections:
                    1. Threadloop before $S_i$
                    2. $S_i$
                    3. Threadloop after $S_i$
                     else
                        perform Loop Fission to the threadloop containing Si
                      endif

                     S ← construct parent of S in the AST (abstract syntax tree)
                        endloop
```

- Note: control flow statements are treated as synchronization statements.
- The following code shows an example of the algorithm. Maroon sections show the change made in every step of the code from the previous one.

Step 1: Original Code + Threadloop

```
__global__ void kernel(){
        body1;
        threadloop1{
                body2;
                for1 (a=a0; a<aend; a++) {
                body3;
                for2 (b=b0; b<bend; b++){
                        body4;
                        Sync1;
                        body5;
                }
                body6;
                if (cond)
                        break;
                body7;
                }
                body8;
        }
        body9;
}
```

Step 2: for → while

```
__global__ void kernel (){
        body1;
        threadloop1{
                body2;
                a = a0;
                while1 (a<aend) {
                body3;
                b = b0;
                while2 (b<bend){
                        body4;
                        Sync1;
                        body5;
                        b ++;
                }
                body6;
                if (cond)
                        break;
                body7;
                a ++;
                }
                body8;
        }
        body9;
}
```

Step 3: Deep Fission (Sync1)

```
__global__ void kernel (){
        body1;
        threadloop1{
                body2;
                a = a0;
                while1 (a<aend) {
                body3;
                b = b0;
                while2 (b<bend){
                        threadloop2{
                        body4;
                        }
        threadloop3{
                        body5;
                        b ++;
                        }
                }
                body6;
                if (cond)
                        break;
                body7;
                a ++;
                }
                body8;
        }
        body9;
}
```

Step 4: Deep Fission (while2)

```
__global__ void kernel (){
        body1;
        threadloop1{
                body2;
                a = a0;
                while1 (a<aend) {
                threadloop4{
                        body3;
                        b = b0;
                }
                while2 (b<bend){
                        threadloop2{
                        body4;
                        }
                        // Sync1;
                        threadloop3{
                        body5;
                        b ++;
                        }
                }
                threadloop5{
                body6;
                if (cond)
                        break;
                body7;
                a ++;
                }
                }
                body8;
        }
        body9;
}
```

Step 5: Loop Fission (while1)

```
__global__ void kernel (){
    body1;
    threadloop1{
        body2;
        a = a0;
    }
    while1 (a<aend) {
        Threadloop4{
            body3;
            b = b0;
        }
        while2 (b<bend){
            threadloop2{
            body4;
        }
        // Sync1;
        threadloop3{
            body5;
            b ++;
        }
        threadloop5{
            body6;
            if (cond)
                break;
            body7;
            a ++;
        }
    }
    threadloop6{
        body8;
    }
    body9;
}
```

Step 6: Loop Fission (break)

```
__global__ void kernel (){
    body1;
    threadloop1{
        body2;
        a = a0;
    }
    while1 (a<aend) {
        Threadloop4{
            body3;
            b = b0;
        }
        while2 (b<bend){
            threadloop2{
            body4;
        }
        // Sync1;
        threadloop3{
            body5;
            b ++;
        }
        threadloop5{
            threadloop7{
                body6;
            }
            if (cond)
                break;
            threadloop8{
                body7;
                a ++;
            }
        }
    }
    threadloop6{
        body8;
    }
    body9;
}
```

Thread Synchronization - Memory Allocation

- Thread private variables need to remain live in all threadloops.
- Universal replication: For each thread-private variable, create a three-dimensional array the size of the RacEr_num_threads_per_core variable.
- Selective replication: Only do so for necessary variables:
    - Kernels with no synchronization statements do not need replication.
    - Variables with live ranges contained inside their thread loop (not accessed outside) do not need to be replicated.

CUDA to RacEr

Replacement Reference

The following table includes all high-level terms in CUDA and their equivalent for the RacEr.

Replacement Reference Guide for CUDA to RacEr Translation

CUDA	RacEr
gridDim <x, y, z>	const int k_gridDim_x const int k_gridDim_y const int k_gridDim_z
blockDim <x, y, z>	const int k_blockDim_x const int k_blockDim_y const int k_blockDim_z
blockIdx <x, y, z>	const int k_blockIdx_x const int k_blockIdx_y const int k_blockIdx_z
threadIdx <x, y, z>	Threadloop iterators (iter_x, iter_y, iter_z)
	#define RACER_TILE_GROUP_<X,Y,Z>_DIM #define RACER_TILE_GROUP_SIZE= RACER_TILE_GROUP_X_DIM × RACER_TILE_GROUP_Y_DIM × RACER_TILE_GROUP_Z_DIM × RacEr_x, RacEr_y, RacEr_z Initialized by bsg_set_tile_x_y() #define k_num_threads (_x, _y, _z) = blockDim / RacEr_TILE_GROUP_<X,Y,Z>_DIM
	-------Memory Allocation------

Thread-private memory
Declaration: int p;
Access: f(p);
	Only if p is alive outside of scope
	Declaration: int p [num_threads_z][num_threads_y]
	[num_threads_z]
	Access: f (p[iter_z][iter_y][iter_x])

Shared memory
Declaration: __shared__ int s[1024];
Access: f(s[idx]);
	Declaration: RacEr_tilegroup_int s (1024) ;
	Access: f(*RacEr_tilegroup_ptr(s, idx))

	-------Synchronization------
__syncthreads __syncthreads_or __syncthreads_and __syncthreads_count	RacEr_tile_group_barrier RacEr_tile_group_barrier_or RacEr_tile_group_barrier_and RacEr_tile_group_barrier_count

Usage

RacEr manycore programs are running in Amazon Web Services (AWS) F1 instance. This infrastructure provides Xilinx Alveo Data Center Card (https://www.xilinx.com/products/boards-and-kits/alveo/u280.html) attached to standard machine instances. We provide pre-configured Linux images and pre-compiled Amazon FPGA Images (AFIs / AGFIs) to reduce setup time. This is a standard build and development environment that is portable across sites. We assume familiarity with AWS EC2 and an existing AWS account.

Software Libraries

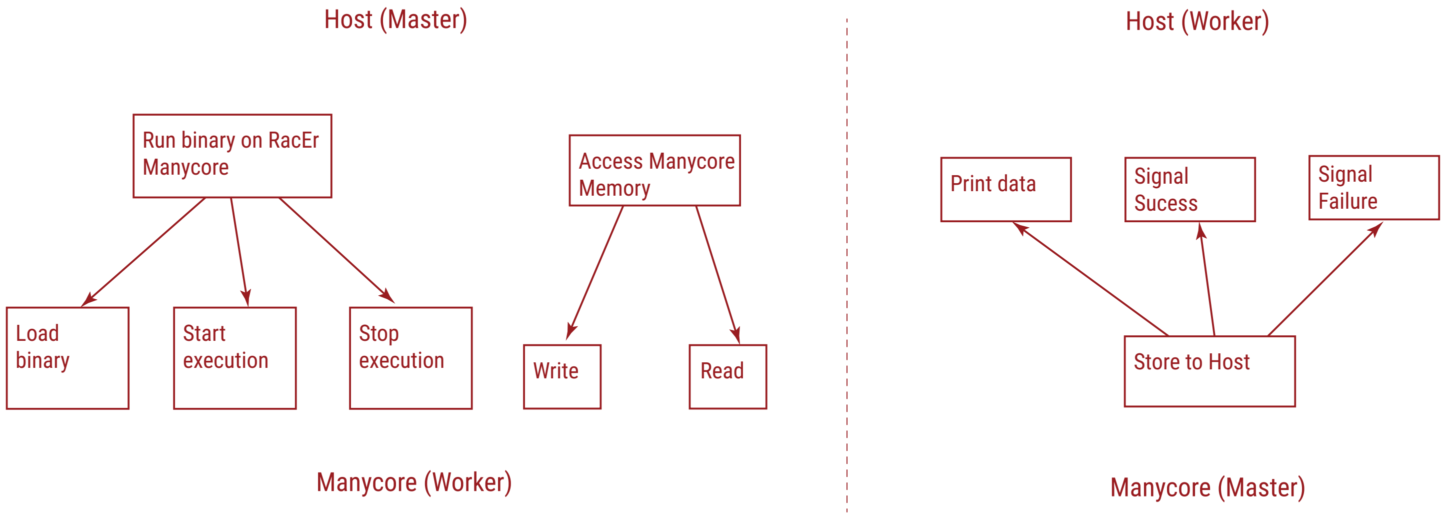
Once a Manycore program has been compiled, it may be loaded onto the Manycore and run. We also provide memory primitives that allow host programs to read from and write to and from the Manycore's memory. Together, these features give the user considerable flexibility in interacting with the Manycore; we conclude with a short explanation on how to write, compile, and run host programs.

Kernel Drivers

RacEr come swith pre-installed kernel drivers. The F1 environment will use a driver developed by AWS. The driver is used to memory map one of the FPGA's Base Address Registers (BARs) so that host programs may access the FPGA from userspace.

User-Level Library

The User-Level Library provides an interface to communicate with the Manycore as either a Master or Worker. The interface's feature set is described in the figure below:



FIFO Interface

All communication between the Host and Manycore occurs over PCIe. PCIe interactions where the Manycore is the Master are routed to a separate FPGA FIFO channel from those where the Manycore is the Worker. Each FIFO is memory mapped to a userspace address. When host programs write and read to these FIFOs through the API functions, the appropriate FIFO is accessed depending on the type of interaction.

Loader

Host program may load a binary such that it is executed by multiple tiles in multiple tile groups. To configure the tiles for execution, a host program must set each tile's tile group and load the binary into the tiles. To set the tile group of a tile, use RacEr_mc_set_tile_group_origin(). To load the binary into the tiles, use RacEr_mc_load_binary(), which accepts the path of the binary as well as a list of (x, y) coordinates to be loaded. The binary's instructions are loaded into each tile's icache and into a predefined segment of DRAM. Additionally, the binary's data segment is loaded into the data memory of each tile. Subsequent to loading, the RacEr_mc_unfreeze() may be used to start a specific tile's execution; RacEr_mc_freeze() may be called to restart a specific tile.

Endpoint Physical Address (EPA) Mem-Copy API

Host programs may also use the master channel to write to and read from the Manycore's memory. To write a buffer on the host to RacEr Manycode memory, use the function RacEr_mc_copy_to_epa(). To read from the the RacEr Manycore, use RacEr_mc_copy_from_epa().

Basic Program Structure

A host program must be written in C or C++ and should first call RacEr_mc_init_host(). This maps the PCIe FIFOs to user space. Once this function is called, the userspace program may interact with the FPGA through the User-Level Library; the headers for the API, which are located in the standard Linux location for header files, should be included.

Compiling and Running

Before running a host program, the appropriate bitstream must be loaded onto the Xilinx Alveo Data Center Card (https://www.xilinx.com/products/boards-and-kits/alveo/u250.html), and the user-level library must be compiled into a dynamic library and moved to the standard location for Linux libraries.

All information is provided as is. There is no warranty that it is correct or suitable for any purpose, neither implicit nor explicit.