Advanced Parallel Architecture

Annalisa Massini - 2014/2015

GPU - Graphics Processing Units

Computer Architecture - A Quantitative Approach, Fifth Edition

Hennessy Patterson

- Chapter 4 - Data-Level Parallelism in Vector, SIMD, and GPU Architectures
  - **Section 4.4 – Graphics Processing Units**

Graphics Processing Units

- GPUs and CPUs do **not** go back in computer architecture genealogy to a **common ancestor**
- The primary ancestors of GPUs are graphics accelerators
- Given the hardware invested to do graphics well architects ask

how can be the design of GPUs used to improve the performance of a wider range of applications?

Graphics Processing Units

- The **challenge** for the GPU programmer
  - is **not** simply **getting good performance** on the GPU
  - but also in **coordinating the scheduling** of **computation** on the system processor and the GPU and the **transfer of data** between system memory and GPU memory

- GPUs have virtually **every type of parallelism** that can be captured by the programming environment:
  - multithreading
  - MIMD
  - SIMD
  - and even instruction-level

Programming the GPU

- NVIDIA developed a ***C-like language and programming environment***: **CUDA - Compute Unified Device Architecture**
- CUDA produces C/C++ for the system processor - ***host*** - and a C and C++ dialect for the GPU - ***device*** (D in CUDA)
- A similar programming language is **OpenCL**, which several companies are developing as vendor-independent language for multiple platforms
- NVIDIA unified all forms of parallelism in the ***CUDA Thread***
- The compiler and the hardware can gang thousands of CUDA Threads together to utilize the various styles of parallelism within a GPU (multithreading, MIMD, SIMD, ILP)

Programming the GPU

- NVIDIA classifies the CUDA programming model as Single Instruction, Multiple Thread (***SIMT***)
- Threads are blocked together - ***Thread Block*** - and executed in **groups of 32** threads
- In Hennessy Patterson the hardware that executes a whole **block of threads** is called a ***multithreaded SIMD Processor***

Programming the GPU

- To distinguish between functions for the GPU (device) and functions for the system processor (host), CUDA uses
  - `__device__` or `__global__` for the device
  - `__host__` for the processor

- CUDA variables declared as in the
  - `__device__` or `__global__` functions

  are allocated to the GPU Memory which is accessible by all multithreaded SIMD processors

Programming the GPU

- The call syntax for the function *name* that runs on the GPU

```
name<<<dimGrid, dimBlock>>>(... parameter list ...)
```

where `dimGrid` and `dimBlock` specify the dimensions of the code (in blocks) and the dimensions of a block (in threads)

- CUDA provides keywords for:
  - the identifier for blocks per grid - **`blockIdx`** - and
  - the identifier for threads per block - **`threadIdx`** -
  - the number of threads per block - **`blockDim`** - which comes from the **`dimBlock`** parameter

Example

- Consider the DAXPY example

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
      for (int i = 0; i < n; ++i)
            y[i] = a*x[i] + y[i];
}
```

Example

- In the CUDA version, we launch:
  - n threads, one per vector element
  - with 256 CUDA Threads per thread block
  - in a multithreaded SIMD Processor

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
      daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      if (i < n) y[i] = a*x[i] + y[i];
}
```

Example

- The GPU function calculates the corresponding element **index i based on** the **block ID**, the **number of threads per block**, and the **thread ID**
- If this index is within the array (i < n), it performs the multiply and add

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
      daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      if (i < n) y[i] = a*x[i] + y[i];
}
```

Threads and Blocks

- The C version has:
  - a loop where each iteration is independent of the others
  - This allows the loop to be transformed straightforwardly into a parallel code
  - **each loop iteration** becomes an **independent thread**
- The programmer determines the parallelism in CUDA explicitly by specifying
  - the grid dimensions and
  - the number of threads per SIMD Processor
- By assigning a single thread to each element, there is **no need to synchronize** among threads when writing results to memory

Threads and Blocks

- A **thread** is associated with **each data element**
  - *CUDA threads*, with thousands of which for various styles of parallelism (multithreading, SIMD, MIMD, ILP)
- Threads are organized into **blocks**
  - ***Thread Blocks***: groups of up to 512 elements
  - ***Multithreaded SIMD Processor***: hardware that executes a whole thread block (32 elements executed per thread at a time)
- Blocks are organized into a **grid**
  - **Blocks are executed independently** and in any order
  - Different **blocks cannot communicate directly** but can *coordinate* using atomic memory operations in GPU Main Memory

Threads and Blocks

- The **programmer** determines the parallelism in CUDA explicitly by specifying the **grid dimensions** and the **number of threads per SIMD Processor**
- By assigning a single thread to each element, there is **no need to synchronize** among threads when writing results to memory
- **GPU hardware** handles **parallel execution** and **thread management** (not done by applications or OS)
  - A multiprocessor composed of multithreaded SIMD processors
  - A Thread Block Scheduler

Threads and Blocks

- Performance programmers must keep the GPU hardware in mind when writing in CUDA they need to
  - **keep groups of 32 threads together** in control flow to get the best performance from multithreaded SIMD Processors
  - **create many more threads** per multithreaded SIMD Processor to hide latency to DRAM.
  - **keep the data addresses localized** in one or a few blocks of memory to get the expected memory performance
- Like many parallel systems, a compromise between productivity and performance is for CUDA to include intrinsics **to give programmers explicit control of the hardware**

NVIDIA GPU Computational Structures

- Similarities between vector architectures and GPU:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Many registers (GPU much more)

- Differences between vector architectures and GPU :
  - No scalar processor (GPU in hw, Vector arch in sw)
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

NVIDIA GPU Computational Structures

- **Efficient code for both** vector architectures and GPUs requires programmers to think in **groups of SIMD operations**

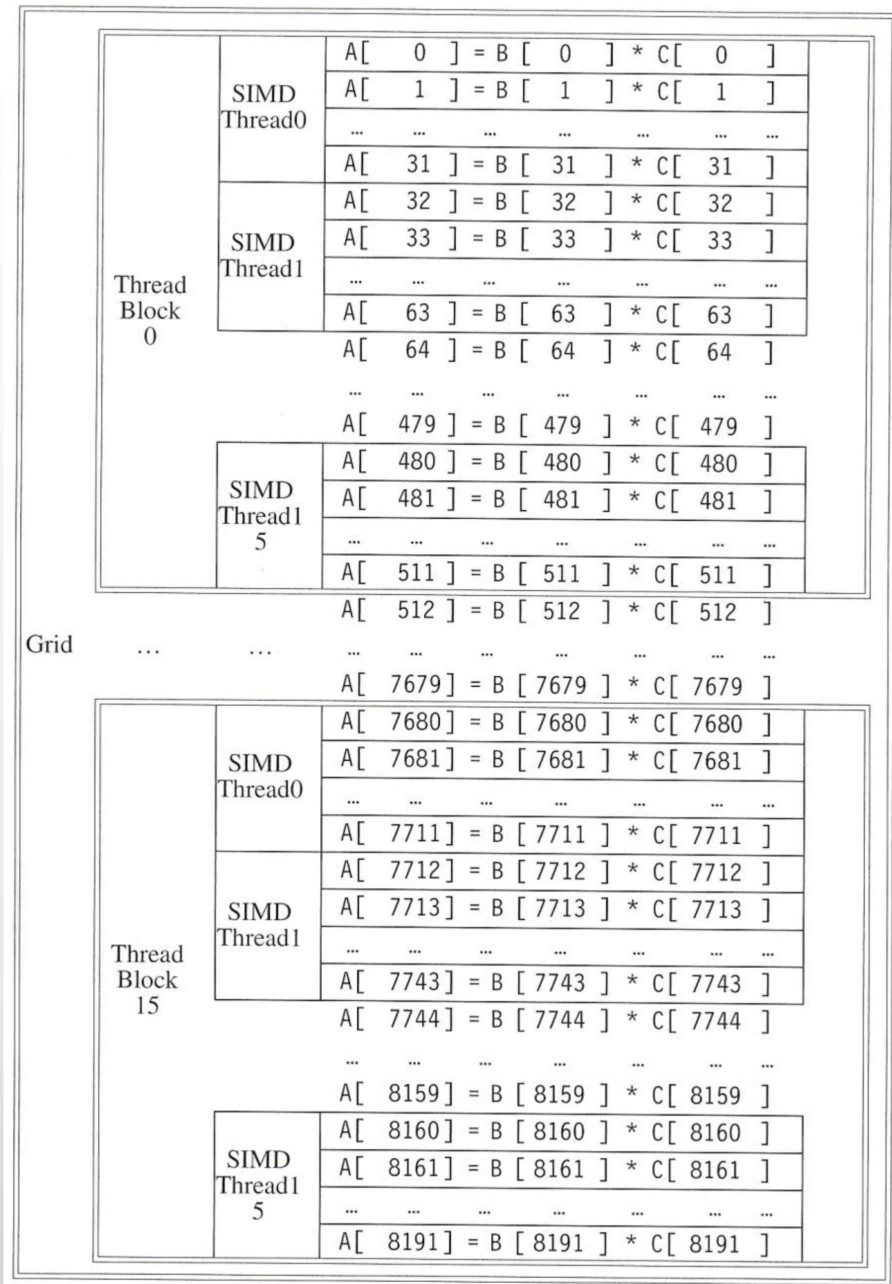
- A *Grid* is the code that runs on a GPU that consists of a set of *Thread Blocks*
- The analogy is :
  - between a **grid** and a **vectorizabled loop** and
  - between a **Thread Block** and the **body of that loop** (strip-mined, for a full computation loop)

Example

- Multiply two vectors of length 8192

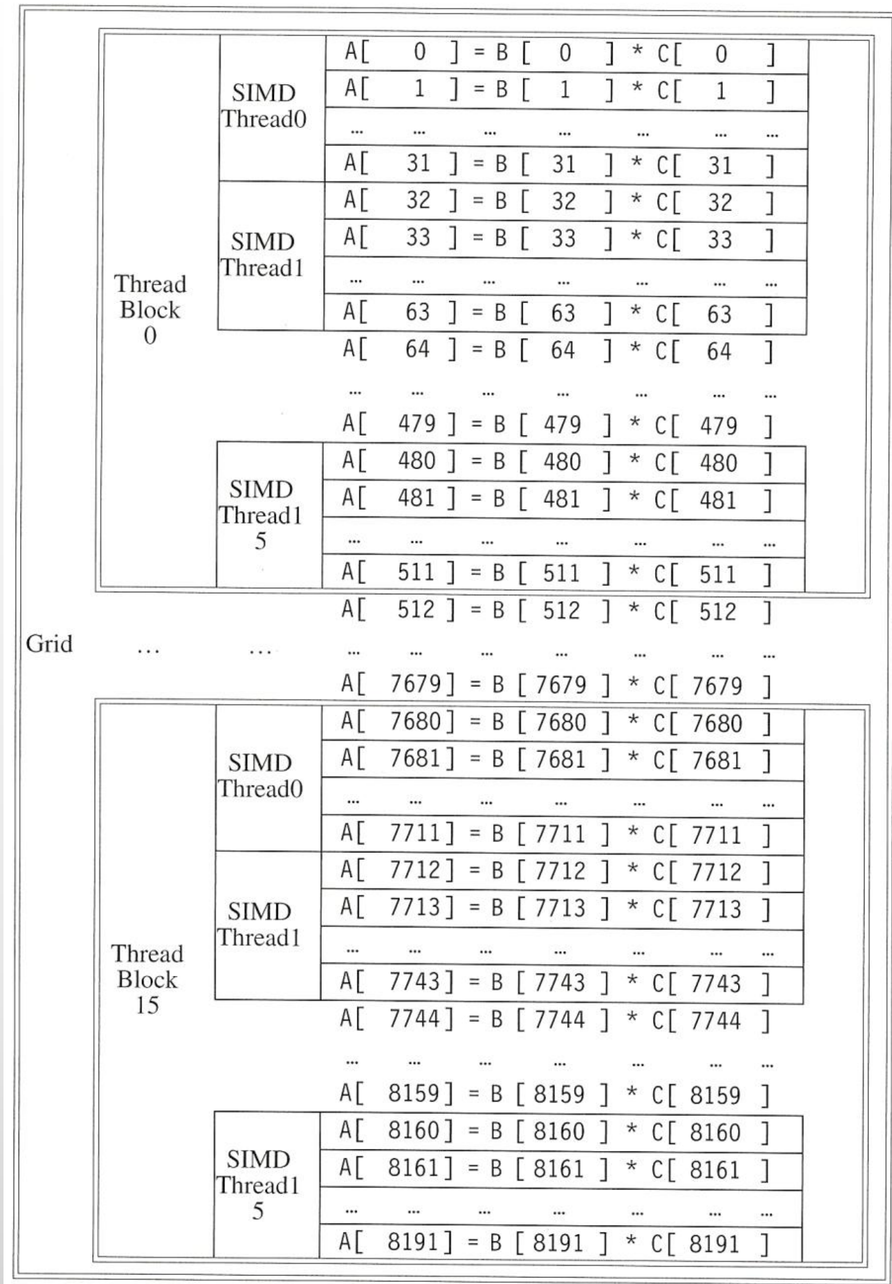
The mapping is:

- **Grid** → vectorizable loop
- **Thread Blocks** → SIMD basic blocks
- **threads of SIMD instructions** → vector-vector multiply



Example

- Multiply two vectors of length 8192
- Code that works over all elements is the **grid**
- Thread blocks break the grid manageable sizes
  - Grid is composed of blocks with up to 512 elements
  - SIMD instruction executes 32 elements at a time
  - The number of Threads Blocks is 8192/512=16



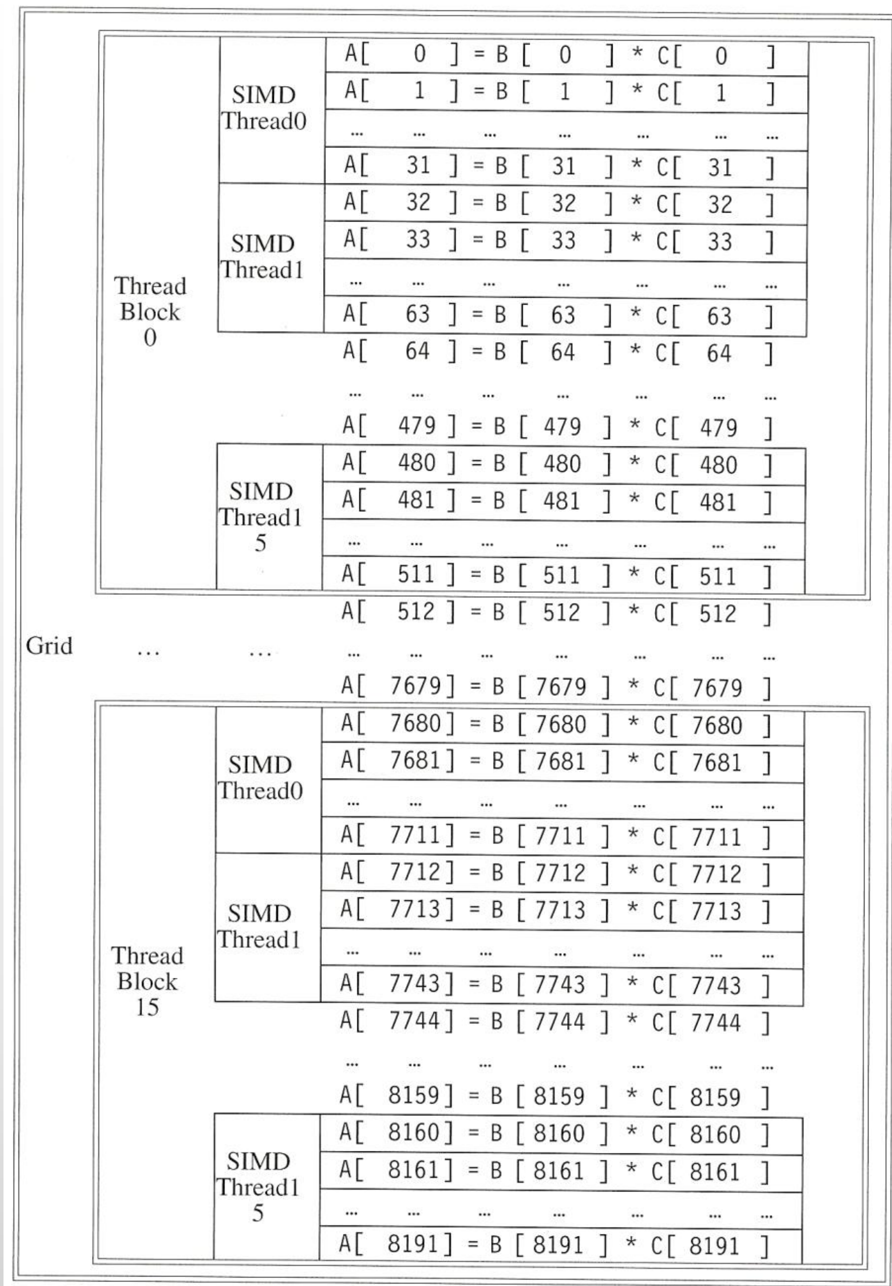
Example

Thread Block:

- analogous to a strip-mined vector loop with vector length of 32
- is assigned to a ***multithreaded SIMD Processor*** by the ***Thread Block Scheduler***

The Thread Block Scheduler:

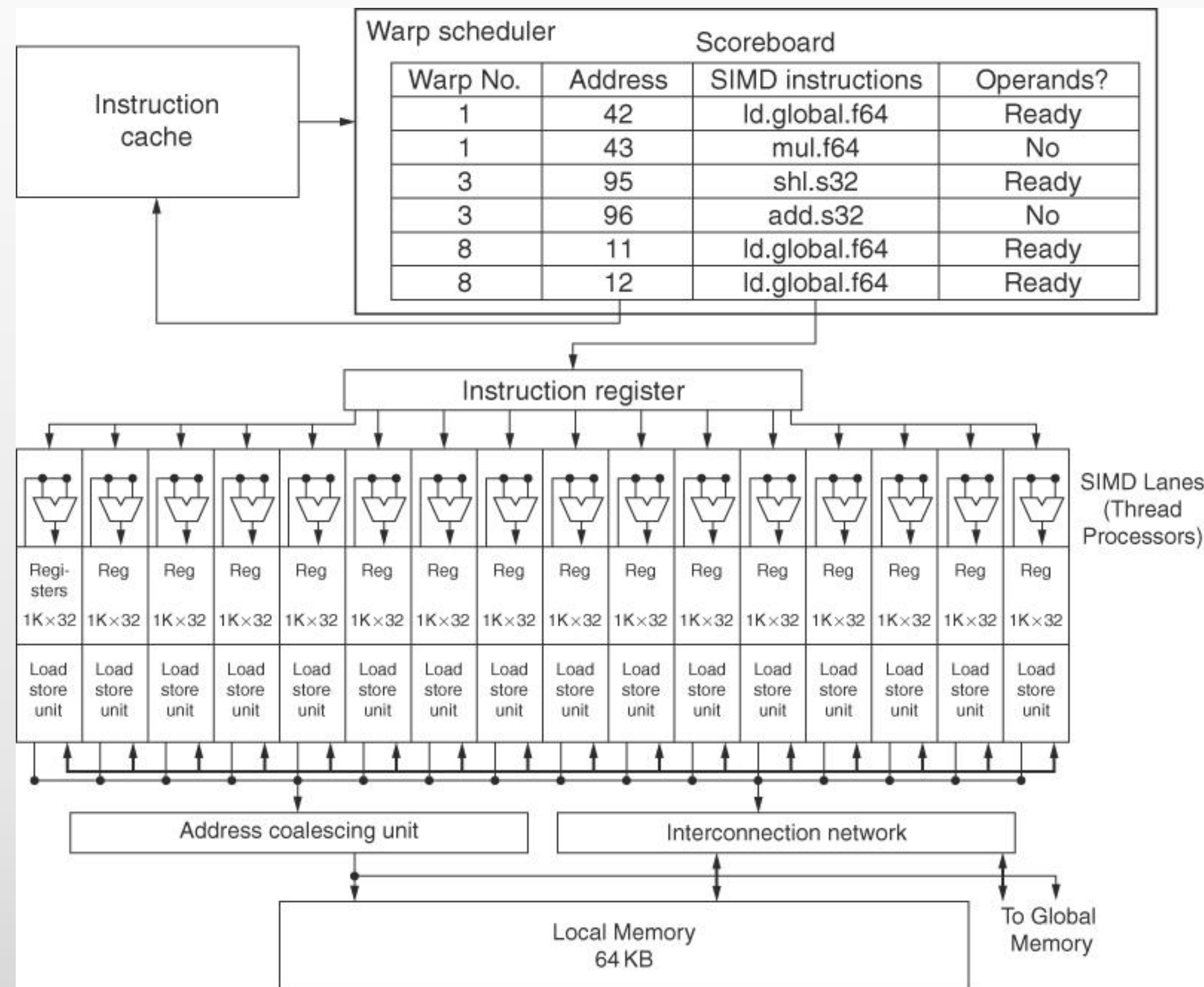
- determines the number of thread blocks needed for the loop and
- keeps allocating them to different multithreaded SIMD Processors until the loop is completed



NVIDIA GPU Computational Structures

A multithreaded SIMD Processor has many **parallel functional units: *SIMD Lanes***

SIMD Lanes are similar to the Vector Lanes, and are much more than a few deeply pipelined in a Vector Processor



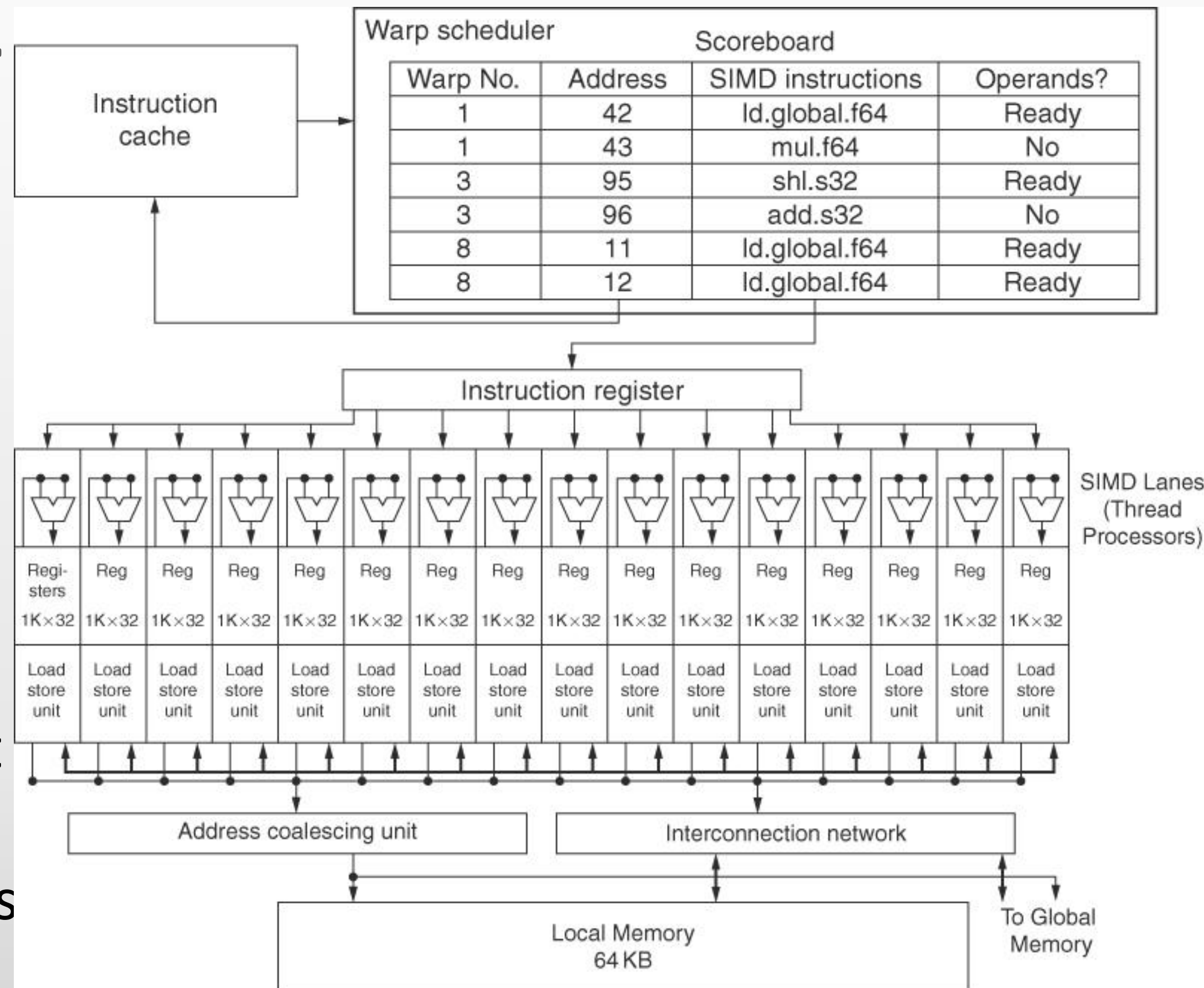
Simplified block diagram of a multithreated SIMD Processor

NVIDIA GPU Computational Structures

GPU - **multiprocessor** composed of **multithreaded SIMD Processors**

The GPU hardware

- contains a collection of multithreaded SIMD Processors that execute a Grid of Thread Blocks (bodies of vectorized loop)

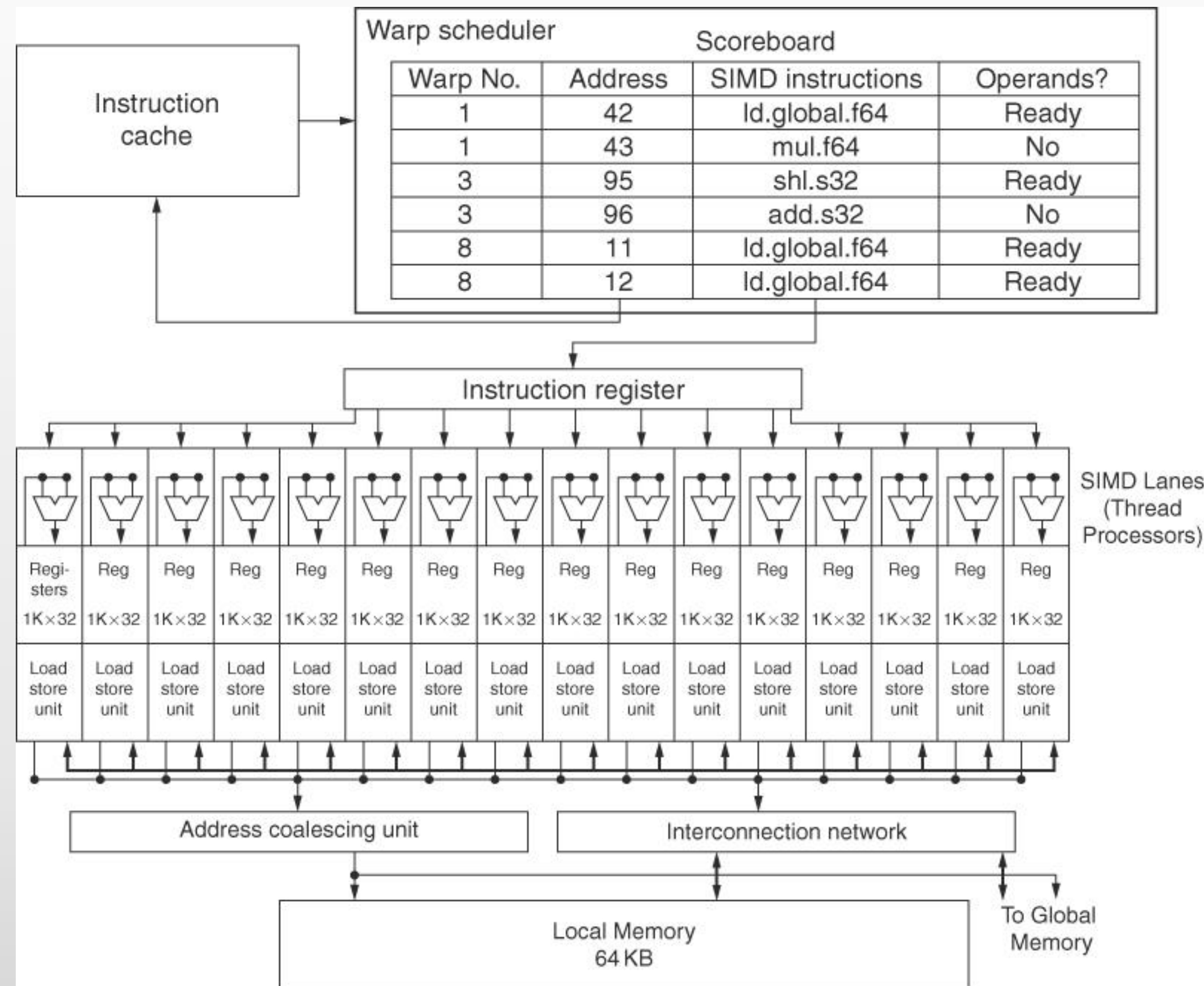


Simplified block diagram of a multithreated SIMD Processor

NVIDIA GPU Computational Structures

Multithreaded SIMD Processors

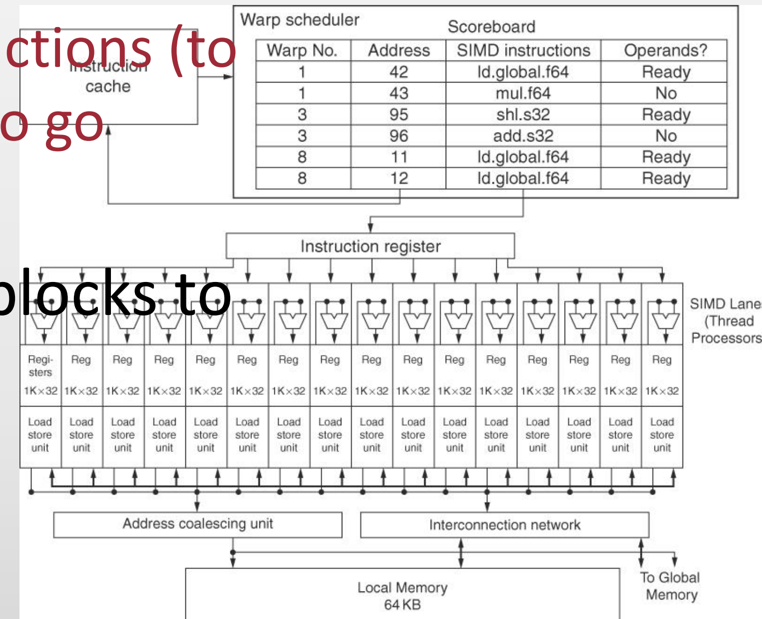
- **full processors** with **separate PCs**
- programmed using threads
- the machine object that the hw creates, manages, schedules, and executes is a *thread of SIMD instructions*



Simplified block diagram of a multithreated SIMD Processor

NVIDIA GPU Computational Structures

- *Threads of SIMD instructions*
  - Each has its own PC
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Keeps track of threads of SIMD instructions (to see which SIMD instruction is ready to go
    - Hides memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - SIMD lanes
  - Wide and shallow compared to vector processors



NVIDIA GPU Computational Structures

- NVIDIA GPU has 32,768 registers
  - Divided into lanes
  - Each SIMD thread is limited to 64 registers
  - SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
- Like a vector processor, these registers are divided logically across the vector lanes → SIMD Lanes in this case
- In the **vector multiply example**, each multithreaded SIMD Proc:
  - must load 32 elements of two vectors from memory into registers
  - perform the multiply by reading and writing registers
  - store the product from registers into memory

NVIDIA Instruction Set Architecture

- Instruction Set target of the NVIDIA compiler is an abstraction of the hardware instruction set
  - "Parallel Thread Execution (PTX)"
  - The hardware instruction set is hidden from the programmer
  - PTX uses virtual registers → compiler figures out how many physical vector registers a SIMD thread needs
  - an optimizer divides the available register storage between the SIMD threads
  - Translation to machine code is performed in software

NVIDIA Instruction Set Architecture

The format of a PTX instruction is

```
opcode.type d, a, b, c;
```

- `d` is the destination operand; `a`, `b`, and `c` are source operands
- The operation type is one of the following:

Type	.type Specifier
Untyped bits 8, 16, 32, and 64 bits	.b8, .b16, .b32, .b64
Unsigned integer 8, 16, 32, and 64 bits	.u8, .u16, .u32, .u64
Signed integer 8, 16, 32, and 64 bits	.s8, .s16, .s32, .s64
Floating Point 16, 32, and 64 bits	.f16, .f32, .f64

NVIDIA Instruction Set Architecture

The groups of instructions are:

- `Arithmetic`
- `Special Functions (mathematical)`
- `Logical`
- `Memory access`
- `Control Flow`
- The control flow instructions are functions `call` and `return`, thread `exit`, `branch`, and barrier synchronization for threads within a thread block (`bar.sync`)

Arithmetic group

```
arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64
add.type        add.f32 d, a, b        d = a + b;
sub.type        sub.f32 d, a, b        d = a – b;
mul.type        mul.f32 d, a, b        d = a * b;
mad.type        mad.f32 d, a, b, c     d = a * b + c;
div.type        div.f32 d, a, b        d = a / b;
rem.type        rem.u32 d, a, b        d = a % b;
abs.type        abs.f32 d, a           d = |a|;
neg.type        neg.f32 d, a           d = 0 - a;
min.type        min.f32 d, a, b        d = (a < b)? a:b;
max.type        max.f32 d, a, b        d = (a > b)? a:b;
setp.cmp.type setp.lt.f32 p, a, b      p = (a < b);
numeric .cmp = eq, ne, lt, le, gt, ge;
unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan
```

ecc.

Special and logic groups

```
special .type = .f32 (some .f64)
rcp.type rcp.f32 d, a d = 1/a; reciprocal
sqrt.type sqrt.f32 d, a d = sqrt(a); square root
rsqrt.type rsqrt.f32 d, a d = 1/sqrt(a); reciprocal square root
sin.type sin.f32 d, a d = sin(a); sine
cos.type cos.f32 d, a d = cos(a); cosine
lg2.type lg2.f32 d, a d = log(a)/log(2) binary logarithm
ex2.type ex2.f32 d, a d = 2 ** a; binary exponential
logic.type = .pred,.b32, .b64
and.type and.b32 d, a, b d = a & b;
or.type or.b32 d, a, b d = a | b;
xor.type xor.b32 d, a, b d = a ^ b;
not.type not.b32 d, a, b d = ~a; one's complement
cnot.type cnot.b32 d, a, b d = (a==0)? 1:0; C logical not
shl.type shl.b32 d, a, b d = a << b; shift left
shr.type shr.s32 d, a, b d = a >> b; shift right
```

Memory access and control flow groups

Memory Access

```
memory.space = .global, .shared, .local, .const
.type = .b8,, .u8, .s8, .b16, .b32, .b64
ld.space.type          ld.global.b32 d, [a+off]       d = *(a+off);
st.space.type          st.shared.b32 [d+off],         a *(d+off) = a;
tex.nd.dtyp.btype      tex.2d.v4.f32.f32 d, a, b      d = tex2d(a, b);
atom.spc.op.type       atom.global.add.u32 d,[a], b
                       atom.global.cas.b32 d,[a], b, c
atom.op = and, or, xor, add, min, max, exch, cas;
.spc = .global; .type = .b32
```

Control Flow

```
branch         @p bra target                 if (p) goto target;
call           call (ret), func, (params)    ret = func(params);
ret            ret                           return;
bar.sync       bar.sync d                    wait for threads
exit           exit                          exit;
```

NVIDIA Instruction Set Arch.

- Sequence of PTX instructions for one iteration of DAXPY loop

```
shl.s32 R8, blockIdx, 9          ; Thread Block ID * Block size (512=2^9)
add.s32 R8, R8, threadIdx        ; R8 = i = my CUDA thread ID
ld.global.f64 RD0, [X+R8]        ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]        ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4            ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2            ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0        ; Y[i] = sum (X[i]*a + Y[i])
```

- CUDA
  - assigns one CUDA Thread to each loop iteration
  - offers a unique identifier number to each thread block (`blockIdx`) and one to each CUDA Thread within a block (`threadIdx`)
  - uses the unique number to address each element in the array
  - 8192 CUDA threads are created!

NVIDIA Instruction Set Arch.

- Sequence of PTX instructions for one iteration of DAXPY loop

```
shl.s32 R8, blockIdx, 9       ; Thread Block ID * Block size (512=2^9)
add.s32 R8, R8, threadIdx     ; R8 = i = my CUDA thread ID
ld.global.f64 RD0, [X+R8]     ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]     ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4         ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2         ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0     ; Y[i] = sum (X[i]*a + Y[i])
```

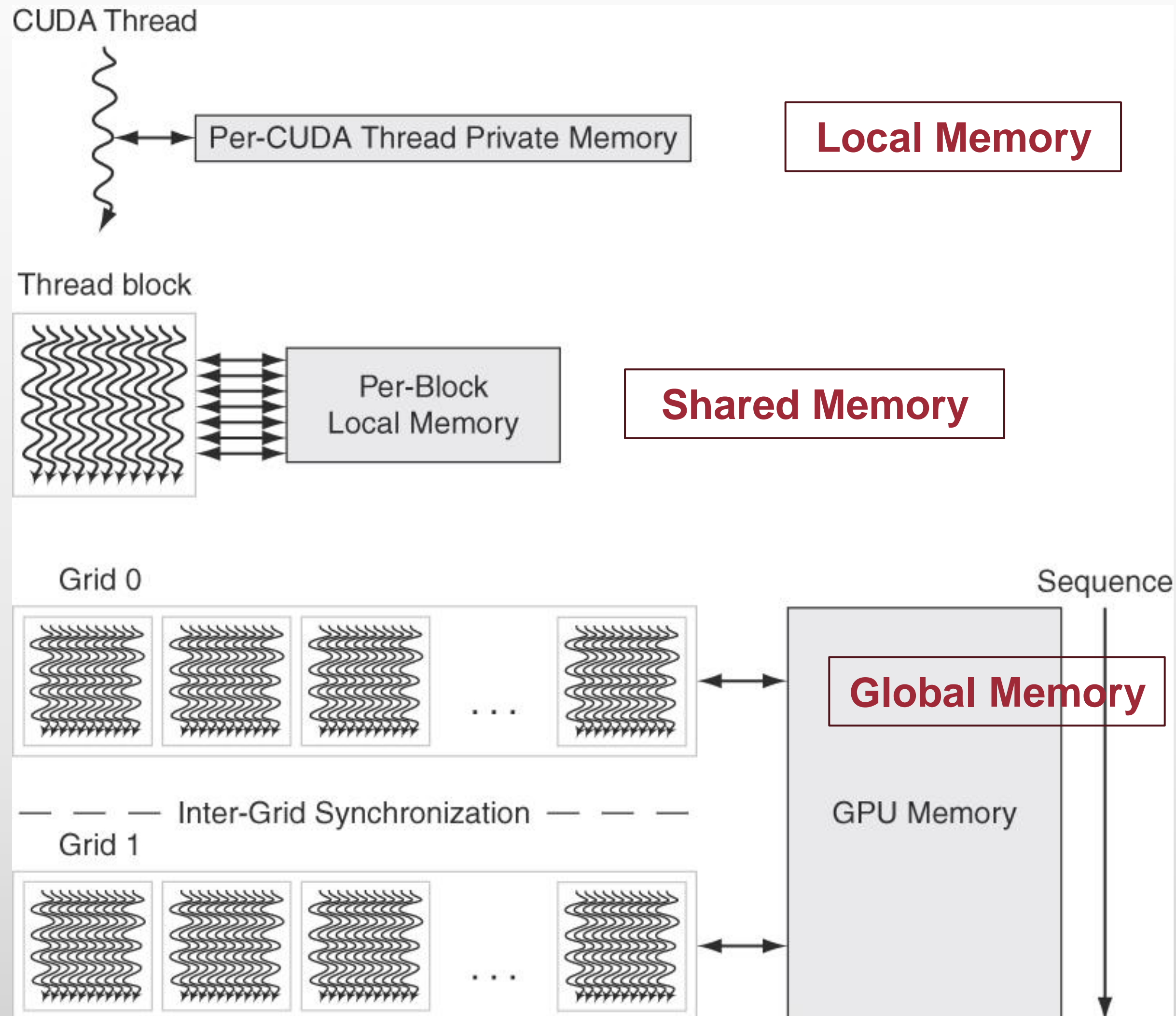
- GPUs
  - All data transfers are gather-scatter!
  - include special Address Coalescing hw to recognize when the SIMD Lanes within a thread are issuing sequential addresses
  - The GPU programmer must ensure that adjacent CUDA Threads access nearby addresses at the same time that can be coalesced, as in our example

NVIDIA GPU Memory Structures

- Each **SIMD Lane** has private section of *off-chip* DRAM
  - "Private memory", not shared by any other lanes
  - Contains stack frame, spilling registers, and private variables
  - Recent GPUs cache this in L1 and L2 caches
- Each **multithreaded SIMD processor** also has local memory that is *on-chip*
  - Shared by SIMD lanes / threads *within a block only*
- The *off-chip* memory shared by SIMD processors is *GPU Memory*
  - Host can read and write GPU memory

NVIDIA GPU Memory Structures

GPU Memory is shared by all Grids (vectorized loops) **Local Memory** is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), **Private Memory** is private to a single CUDA Thread.



Summary

- GPUs are really just multithreaded SIMD processors, although they have:
  - more processors,
  - more lanes per processor, and
  - more multithreading hardware than traditional multicore computers
- The CUDA programmer can think of programming **thousands of threads**, although they are really executing each **block of 32 threads** on the **many lanes** of the many SIMD Processors
- The CUDA programmer who wants good performance keeps in mind that these **threads are blocked and executed 32 at a time** and that **addresses need to be to adjacent** addresses to get good performance from the memory system