EECS 570
Lecture 4
GPUs
Winter 2022
Prof. Yatin Manerkar
http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Martin, Roth, Nowatzyk, Wenisch, of EPFL, CMU, UPenn, U-M, UIUC.
• Slides developed in part by Profs. Adve, Falsafi, Martin, Roth, Nowatzyk, and Wenisch of EPFL, CMU, UPenn, U-M, UIUC.
Readings

This Wednesday:

• Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Ch. 3.1-3.3, 4.1-4.3

Using Vectors in Your Code
Using Vectors in Your Code

• Write in assembly
  □ Ugh

• Use “intrinsic” functions and data types
  □ For example: _mm_mul_ps() and “__m128” datatype

• Use vector data types
  □ typedef double v2df __attribute__((vector_size(16)));

• Use a library someone else wrote
  □ Let them do the hard work
  □ Matrix and linear algebra packages

• Let the compiler do it (automatic vectorization)
  □ GCC’s “-ftree-vectorize” option, -ftree-vectorizer-verbose=n
  □ Limited impact for C/C++ code (old, hard problem)
SAXPY Example: Best Case

• Code

```c
void saxpy(float* x, float* y, float* z, float a, int length) {
    for (int i = 0; i < length; i++) {
        z[i] = a*x[i] + y[i];
    }
}
```

• Scalar

```
.L3:
movss (%rdi,%rax), %xmm1
mulss %xmm0, %xmm1
addss (%rsi,%rax), %xmm1
movss %xmm1, (%rdx,%rax)
addq $4, %rax
cmpq %rcx, %rax
jne .L3
```

• Auto Vectorized

```
.L6:
movaps (%rdi,%rax), %xmm1
mulps %xmm2, %xmm1
addps (%rsi,%rax), %xmm1
movaps %xmm1, (%rdx,%rax)
addq $16, %rax
incl %r8d
cmpl %r8d, %r9d
ja .L6
```

+ Scalar loop to handle last few iterations (if length % 4 != 0)

“mulps”: multiply packed ‘single’
SAXPY Example: Actual

- **Code**
  ```c
  void saxpy(float* x, float* y, float* z, float a, int length) {
      for (int i = 0; i < length; i++) {
          z[i] = a*x[i] + y[i];
      }
  }
  ```

- **Scalar**
  `.L3:
  ```
  movss (%rdi,%rax), %xmm1
  mulss %xmm0, %xmm1
  addss (%rsi,%rax), %xmm1
  movss %xmm1, (%rdx,%rax)
  addq $4, %rax
  cmpq %rcx, %rax
  jne .L3
  ```

  `.L8:
  ```
  movaps %xmm3, %xmm1
  movaps %xmm3, %xmm2
  movlps (%rdi,%rax), %xmm1
  movlps (%rsi,%rax), %xmm2
  movhps 8(%rdi,%rax), %xmm1
  movhps 8(%rsi,%rax), %xmm2
  mulps %xmm4, %xmm1
  incl %r8d
  addps %xmm2, %xmm1
  movaps %xmm1, (%rdx,%rax)
  addq $16, %rax
  cmpl %r9d, %r8d
  jb .L8
  ```

  + Explicit alignment test
  + Explicit aliasing test
Bridging “Best Case” and “Actual”

• Align arrays
  ```c
  typedef float afloat __attribute__((__aligned__(16)));
  void saxpy(afloat* x,
             afloat* y,
             afloat* z,
             float a, int length) {
    for (int i = 0; i < length; i++) {
      z[i] = a*x[i] + y[i];
    }
  }
  ```

• Avoid aliasing check
  ```c
  typedef float afloat __attribute__((__aligned__(16)));
  void saxpy(afloat* __restrict__ x,
             __restrict__ y,
             __restrict__ z, float a, int length)
  ```

• Even with both, still has the “last few iterations” code
• Code

```c
void saxpy(float* x, float* y, float* z, float a, int length) {
    for (int i = 0; i < length; i++) {
        z[i] = a*x[i] + y[i];
    }
}
```

• Scalar

.L3:

```assembly
    movss (%rdi,%rax), %xmm1
    mulss %xmm0, %xmm1
    addss (%rsi,%rax), %xmm1
    movss %xmm1, (%rdx,%rax)
    addq $4, %rax
    cmpq %rcx, %rax
    jne .L3
```

• Auto Vectorized

.L7:

```assembly
    movaps (%rdi,%rax), %xmm0
    incl %ecx
    subps (%rsi,%rax), %xmm0
    addq $16, %rax
    addps %xmm0, %xmm1
    cmpl %ecx, %r8d
    ja .L7
    haddps %xmm1, %xmm1
    haddps %xmm1, %xmm1
    movaps %xmm1, %xmm0
    je .L3
```

- “haddps”: Packed Single-FP Horizontal Add

SSE2 on Pentium 4
Tomorrow’s “CPU” Vectors
Beyond Today’s Vectors

• Today’s vectors are limited
  □ Wide compute
  □ Wide load/store of consecutive addresses
  □ Allows for “SOA” (structures of arrays) style parallelism

• Looking forward (and backward)...
  □ Vector masks
    ▪ Conditional execution on a per-element basis
    ▪ Allows vectorization of conditionals
  □ Scatter/gather
    ▪ $a[i] = b[y[i]]$  $b[y[i]] = a[i]$
    ▪ Helps with sparse matrices, “AOS” (array of structures) parallelism

• Together, enables a different style vectorization
  □ Translate arbitrary (parallel) loop bodies into vectorized code (later)
Vector Masks (Predication)

• **Vector Masks**: 1 bit per vector element
  - Implicit predicate in all vector operations
    
      ```
      for (I=0; I<N; I++) if (maskI) { vop... }
      ```
  - Usually stored in a “scalar” register (up to 64-bits)
  - Used to vectorize loops with conditionals in them
    
    ```
    cmp_eq.v, cmp_lt.v, etc.: sets vector predicates
    ```

    ```
    for (I=0; I<32; I++)
      if (X[I] != 0.0) Z[I] = A/X[I];
    ```

    ```
    ldf.v [X+r1] -> v1
    cmp_ne.v v1,f0 -> r2       // 0.0 is in f0
    divf.sv {r2} v1,f1 -> v2   // A is in f1
    stf.v {r2} v2 -> [Z+r1]
    ```
Scatter Stores & Gather Loads

• How to vectorize:

```c
for(int i = 1, i<N, i++) {
    int bucket = val[i] / scalefactor;
    found[bucket] = 1;
}
```

□ Easy to vectorize the divide, but what about the load/store?

• Solution: hardware support for vector “scatter stores”
  □ \( \text{stf.} \text{v v2->[r1+v1]} \)
  □ Each address calculated from \( r1+v1_i \)
    
    \[ \text{stf v2}_0->[r1+v1_0], \quad \text{stf v2}_1->[r1+v1_1], \]
    \[ \text{stf v2}_2->[r1+v1_2], \quad \text{stf v2}_3->[r1+v1_3] \]

• Vector “gather loads” defined analogously
  □ \( \text{ldf.} \text{v [r1+v1]}->\text{v2} \)

• Scatter/gathers slower than regular vector load/store ops
  □ Still provides throughput advantage over non-vector version
Today’s GPU’s “SIMT” Model
Graphics Processing Units (GPU)

- Killer app for parallelism: graphics (3D games)

- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until recently, programmed through graphics API

- GPU in every desktop, laptop, mobile device
  - massive volume and potential impact

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009 ECE 498AI, University of Illinois, Urbana-Champaign
What is Behind such an Evolution?

• The GPU is specialized for compute-intensive, highly data parallel computation (exactly what graphics rendering is about)
  □ So, more transistors can be devoted to data processing rather than data caching and flow control

• The fast-growing video game industry exerts strong economic pressure that forces constant innovation
GPUs and SIMD/Vector Data Parallelism

- Graphics processing units (GPUs)
  - How do they have such high peak FLOPS?
  - Ans: exploit massive data parallelism

- “SIMT” execution model
  - Single instruction multiple threads
  - Similar to both “vectors” and “SIMD”
  - A key difference: better support for conditional control flow

- Program it with CUDA or OpenCL (or Vulkan or Metal or …)
  - Extensions to C (or Objective-C in the case of Metal)
  - Perform a “shader task” (a snippet of scalar computation) over many elements
  - Internally, GPU uses scatter/gather and vector mask operations
"GPGPU"
- Originally could only perform “shader” computations on images
- So, programmers started using this framework for computation
- Puzzle to work around the limitations, unlock the raw potential

As GPU designers notice this trend...
- Hardware provided more “hooks” for computation
- Provided some limited software tools

GPU designs are now fully embracing compute
- More programmability features to each generation
- Industrial-strength tools, documentation, tutorials, etc.
- Can be used for in-game physics, etc.
- A major initiative to push GPUs beyond graphics (HPC, ML)
• NVIDIA G80 – extreme SIMD parallelism in shader units
Throughput Computing: Hardware Basics

Justin Hensley
Advanced Micro Devices, Inc
Graphics Product Group
A Simple Program - Diffuse Shader

```c
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;
float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

Each invocation is independent, but no explicitly exposed parallelism
Shader is compiled

1 Unshaded fragment in

```c
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;
float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

1 Shaded fragment out

```c
<diffuseShader>:
    sample r0, v4, t0, s0
    mul r3, v0, cb0[0]
    madd r3, v1, cb0[1], r3
    madd r3, v2, cb0[2], r3
    clmp r3, r3, 1(0.0), 1(1.0)
    mul c0, r0, r3
    mul c1, r1, r3
    mul c2, r2, r3
    mov c3, 1(1.0a)
```
Exploit data parallelism! - add two cores

Each invocation is independent!
Add even more cores - four cores
How about even more cores - 16 cores
128 cores?

How do you feed all these cores?

Think data parallel! - Graphics requires hardware process *lots* of “items” that share the same shader
Back to the simple core...

- How do you feed all these cores?
- Share cost of fetch / decode across many ALUs
- SIMD Processing
Back to the simple core...

- How do you feed all these cores?
- Share cost of fetch / decode across many ALUs
- SIMD Processing
  - Single
  - Instruction
  - Multiple
  - Data
Back to the simple core...

- How do you feed all these cores?
- Share cost of fetch / decode across many ALUs
- SIMD Processing
  - Single

SIMD Processing does not imply SIMD instructions!

adapted from Kayvon Fatahalian’s SIGGRAPH'08 talk
<diffuseShader>:
  sample r0, v4, t0, s0
  mul r3, v0, cb0[0]
  madd r3, v1, cb0[1], r3
  madd r3, v2, cb0[2], r3
  clmp r3, r3, l(0.0), l(1.0)
  mul o0, r0, r3
  mul o1, r1, r3
  mul o2, r2, r3
  mov c3, l(1.0)
128-Fragments in parallel

16 cores ➔ 128 ALUs (16 cores * 8 ALUs)
➔ 16 independent instruction streams
128-things in parallel

• X cores can work on primitives (triangles)
  – “geometry shader”
• Y cores can work on vertices
  – “vertex shader”
• Z cores can work on fragments
  – “pixel shader”
• N cores can work on data/work/etc
  – “compute kernels”/“compute shaders”
• Which cores working on what data changes over time
What about branching?

```
<unconditional shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional shader code>
```
What about branching?

```
<unconditional shader code>
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional shader code>
```
What about branching?

Not all ALUs do useful work!
Worst case: 1/8 performance

```c
<unconditional shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional shader code>
```
What about branching?

```
<unconditional shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional shader code>
```
How to handle stalls?

- Memory access latency = 100’s to 1000’s of cycles
  - Stalls occur when a core cannot run the next instruction

- GPUs don’t have the large / fancy caches and logic that helps avoid stall because of a dependency on a previous operation.

- But we have LOTS of independent fragments.
  - Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8

Fetch/Decode

ALU ALU ALU ALU

ALU ALU ALU ALU

Ctx Ctx Ctx Ctx

Ctx Ctx Ctx Ctx

Shared Ctx Data

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8

Frag 9 ... 16

Frag 17 ... 24

Frag 25 ... 32

Fetch/Decode

ALU

ALU

ALU

ALU

ALU

ALU

ALU

ALU

1

2

3

4

31  adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8

Frag 9 ... 16

Frag 17 ... 24

Frag 25 ... 32

Stall

Runnable

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8

Frag 9 ... 16

Frag 17 ... 24

Frag 25 ... 32

1

2

3

4

Stall

Runnable

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

<table>
<thead>
<tr>
<th>Time (clocks)</th>
<th>Frag 1...8</th>
<th>Frag 9...16</th>
<th>Frag 17...24</th>
<th>Frag 25...32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Stall</td>
<td>Stall</td>
<td>Stall</td>
</tr>
<tr>
<td></td>
<td>Runnable</td>
<td></td>
<td>Runnable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
<td>Stall</td>
</tr>
<tr>
<td></td>
<td>Runnable</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Increase run time of one group
To maximum throughput of many groups
Latency Hiding with “Thread Warps”

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- No OS context switching
- Memory latency hiding
  - Graphics has millions of pixels
Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Lock step
  - Programming model is SIMD (no threads) \(\rightarrow\) SW needs to know vector length
  - ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Each thread can be treated individually (i.e., placed in a different warp) \(\rightarrow\) programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar \(\rightarrow\) vector instructions formed dynamically
CUDA In One Slide

Thread

per-thread local memory

Block

Local barrier

per-block shared memory

Kernel foo()

Kernel bar()

Global barrier

per-device global memory

© NVIDIA Corporation 2009
CUDA Devices and Threads

• A compute device
  □ Is a coprocessor to the CPU or host
  □ Has its own DRAM (device memory)
  □ Runs many threads in parallel
  □ Is typically a GPU but can also be another type of parallel processing device

• Data-parallel portions of an application are expressed as device kernels which run on many threads

• Differences between GPU and CPU threads
  □ GPU threads are extremely lightweight
    ○ Very little creation overhead
  □ GPU needs 1000s of threads for full efficiency
    ○ Multi-core CPU needs only a few
Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
Execution Model

• Each thread block is executed by a single multiprocessor
  □ Synchronized using shared memory

• Many thread blocks are assigned to a single multiprocessor
  □ Executed concurrently in a time-sharing fashion
  □ Keep GPU as busy as possible

• Running many threads in parallel can hide DRAM memory latency
  □ Global memory access : 2~300 cycles
CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories
Example: Vector Addition Kernel

// Pair-wise addition of vector elements
// One thread per addition

__global__ void
vectorAdd(float* iA, float* iB, float* oC)
{
    int idx = threadIdx.x
              + blockDim.x * blockIdx.x;
    oC[idx] = iA[idx] + iB[idx];
}

Courtesy NVIDIA
Example: Vector Addition Host Code

```c
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
// ... initialize h_A and h_B

// allocate device memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float) );
cudaMalloc( (void**) &d_B, N * sizeof(float) );
cudaMalloc( (void**) &d_C, N * sizeof(float) );

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice );

// execute the kernel on N/256 blocks of 256 threads each
vectorAdd<<< N/256, 256 >>>( d_A, d_B, d_C);
```

Courtesy NVIDIA
CUDA-Strengths

- Easy to program (small learning curve)

- Success with several complex applications
  - At least 7X faster than CPU stand-alone implementations

- Allows us to read and write data at any location in the device memory

- More fast memory close to the processors (registers + shared memory)
CUDA-Limitations

• Some hardwired graphic components are hidden

• Better tools are needed
  - Profiling
  - Memory blocking and layout
  - Binary Translation

• Difficult to find optimal values for CUDA execution parameters
  - Number of thread per block
  - Dimension and orientation of blocks and grid
  - Use of on-chip memory resources including registers and shared memory

• Working with GPUs is an active area of research
Acknowledgements

Some slides/material from:

- Nikos Hardavellas at Northwestern
- UToronto course by Andreas Moshovos
- UIUC course by Wen-Mei Hwu and David Kirk
- UCSB course by Andrea Di Blas
- Universitat Jena by Waqar Saleem
- NVIDIA by Simon Green and many others
- Real World Technologies by David Kanter
- Tyler Sorensen (UCSC)
GeForce GTX 680 – Streaming Processor Array (SPA)

- 3.54bn transistors
- 1536 CUDA cores
- Core: 1.006GHz – Up to 1.1GHz
- Mem: 6.008GHz (?) – Lab machines report 3.004GHz
- 3.09 TFLOPs
- 195W
GeForce GTX 680 – Graphics Processing Cluster
GeForce GTX 680 – Streaming Multiprocessor

- **SM (a.k.a. SMX, SMP)**
  - Streaming Multiprocessor
  - Multi-threaded processor
    - 192 CUDA cores
    - 1 to 2048 threads active
    - Shared instruction fetch per 32 threads
    - Fundamental processing unit for CUDA thread block

- **SP (a.k.a. CUDA core)**
  - Streaming Processor
  - Scalar ALU for a single CUDA thread

- **SFU**
  - Special function unit

- **LDST**
  - Memory access unit
Scheduling Threads for Execution

• Break data into Blocks (grid)
• Break Blocks into Warps
  – 32 consecutive threads
• Allocate Resources
  – Registers, Shared Mem, Barriers
• Then allocate for execution
- Grid is launched on the SPA
- Kepler allows up to 32-way grid concurrency (streams)
  - GTX680: up to 16 grids

- Thread Blocks are serially distributed to all the SMs
  - Potentially >1 Thread Block per SM

- Each SM launches Warps of 32 Threads
  - 3 levels of parallelism

- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks
Stream Multiprocessors Execute Blocks

- Threads are assigned to SMs at Block granularity
  - Up to **16 Blocks** per SM
  - Up to **64 Resident Warps** per SM
  - Up to **2K threads** per SM
    - Could be 512 (threads/block) * 4 blocks
    - Or 256 (threads/block) * 8 blocks, etc.
  - NOTE: actual # as resources allow

- Threads run concurrently
  - SM assigns/maintains thread id #s
  - SM manages/schedules thread execution

*All numbers are for GTX680 (3.0 capability)*

More info on limits at:
Thread Scheduling and Execution

- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model

- Warp: primitive scheduling unit

- All threads in warp:
  - same instruction
  - control flow causes some to become inactive
  - Up to 512M instructions per kernel
SM hardware implements zero-overhead Warp scheduling

- Scheduler masks out ineligible warps
  - e.g., operands not ready
- Select warp to schedule next based on a prioritized scheduling policy
- Decode instruction
- Issue instruction
- All threads in a Warp execute the same instruction when selected
• Fetch one warp instruction/cycle
  – from instruction L1 cache
  – into any instruction buffer slot
• Issue one “ready-to-go” warp instruction/cycle
  – from any warp - instruction buffer slot
  – operand **scoreboarding** used to prevent hazards
• Issue selection based on round-robin/age of warp: not public
• SM broadcasts the same instruction to 32 Threads of a Warp
• That’s the theory → warp scheduling may use heuristics
Scoreboarding

- How to determine if a thread is ready to execute?
- A **scoreboard** is a table in hardware that tracks
  - instructions being fetched, issued, executed
  - resources they need (functional units and operands)
  - which instructions modify which registers
- Old concept from CDC 6600 (1960s) to separate memory and computation
Scoreboarding

• All register operands of all instructions in the Instruction Buffer are scoreboarded
  – Status becomes ready after the needed values are deposited
  – prevents hazards
  – cleared instructions are eligible for issue

• Decoupled Memory/Processor pipelines
  – any thread can continue to issue instructions until scoreboarding prevents issue
  – allows Memory/Processor ops to proceed in shadow of Memory/Processor ops
Consider three separate instruction streams: warp1, warp3 and warp8

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>42</td>
<td>Computing</td>
</tr>
<tr>
<td>Warp 3</td>
<td>95</td>
<td>Waiting</td>
</tr>
<tr>
<td>Warp 8</td>
<td>11</td>
<td>Operands ready to go</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>Schedule at time k</td>
</tr>
</tbody>
</table>
Consider three separate instruction streams: warp1, warp3 and warp8.

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>42</td>
<td>Ready to write result</td>
</tr>
<tr>
<td>Warp 3</td>
<td>95</td>
<td>Waiting</td>
</tr>
<tr>
<td>Warp 8</td>
<td>11</td>
<td>Computing</td>
</tr>
</tbody>
</table>

Schedule at time k+1