King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 14: GPU Compute APIs 3, GPU Texturing 1

Markus Hadwiger, KAUST

Reading Assignment #8 (until Oct 26)

Read (required):

- Programming Massively Parallel Processors book, 3rd edition, Chapter 7 (*Parallel Patterns: Convolution*)
- Interpolation for Polygon Texture Mapping and Shading, Paul Heckbert and Henry Moreton

`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.7886`

Quiz #2: Oct 28

Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assigments
- Programming assignments (algorithms, methods)
- Solve short practical examples

Semester Project (proposal until next week!)

- Choosing your own topic encouraged!
  (we can also suggest some topics)
  - Pick something that you think is really cool!
  - Can be completely graphics or completely computation, or both combined
  - Can be built on CS380 frameworks, NVIDIA OpenGL SDK, or CUDA SDK
- Submit short (1-2 pages) project proposal sometime next week
- Submit semester project and report (deadline: Dec 10)
- Present semester project (we will schedule event in final exams week)

Semester Project Ideas (1)

Some ideas for topics

- Procedural shading with noise + marble etc. (GPU Gems 2, chapter 26)
- Procedural shading with noise + bump mapping (GPU Gems 2, chapter 26)
- Subdivision surfaces (GPU Gems 2, chapter 7)
- Ambient occlusion, screen space ambient occlusion
- Shadow mapping, hard shadows, soft shadows
- Deferred shading
- Particle system rendering + CUDA particle sort
- Advanced image filters: fast bilateral filtering, Gaussian kD trees
- Advanced image de-convolution (e.g., convex L1 optimization)
- PDE solvers (e.g., anisotropic diffusion filtering, 2D level set segmentation, 2D fluid flow)

Semester Project Ideas (2)

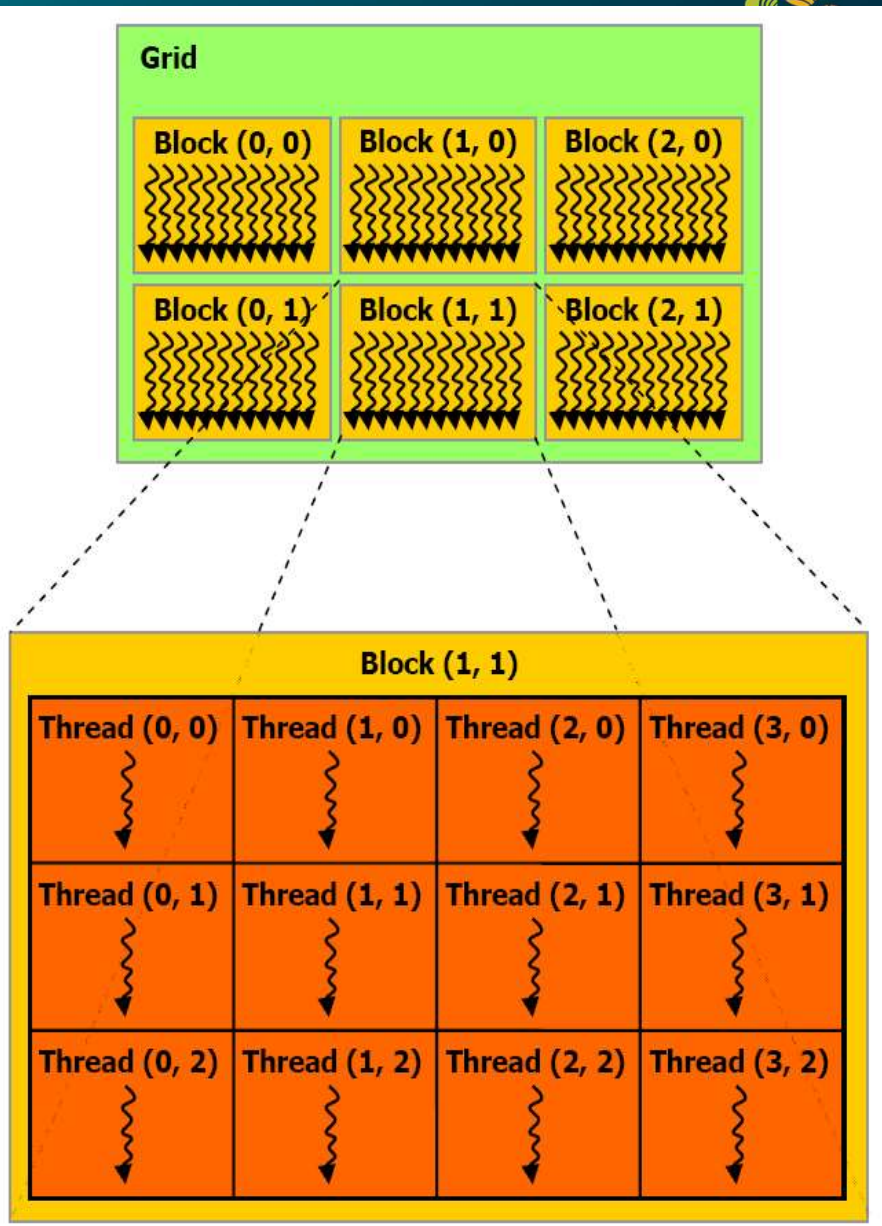
Some ideas for topics

- Distance field computation (GPU Gems 3, chapter 34)
- Livewire (“intelligent scissors“) segmentation in CUDA
- Linear systems solvers, matrix factorization (Cholesky, ...); with/without CUBLAS
- CUDA + matlab
- Fractals (Sierpinski, Koch, ...)
- Image compression
- Bilateral grid filtering for multichannel images
- Discrete wavelet transforms
- Fast histogram computations
- Terrain rendering from height map images; clipmaps or adaptive tesselation

CUDA Multi-Threading

- CUDA model groups threads into blocks; blocks into grid

- Execution on actual hardware:
  - Block assigned to SM (up to 8, 16, or 32 blocks per SM; depending on compute capability)
  - 32 threads grouped into warp



Thread Cooperation

- The Missing Piece: threads may need to cooperate
- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Drastic bandwidth reduction
- Thread cooperation is a powerful feature of CUDA
- Cooperation between a monolithic array of threads is not scalable
  - Cooperation within smaller batches of threads is scalable

Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared
- Allowed in conditional code only if the conditional is uniform across the entire thread block

Synchronization

- Threads in the same block can communicate using shared memory

- __syncthreads()
  - Barrier for threads only within the current block
- __threadfence()
  - Flushes global memory writes to make them visible to all threads

Plus newer sync functions, e.g., from compute capability 2.x: __syncthreads_count(), __syncthreads_and/or(), __threadfence_block(), __threadfence_system(), …

Now: *Must* use versions with _sync suffix, because of Independent Thread Scheduling (compute capability 7.x and newer)!

COOPERATIVE GROUPS

Kyrylo Perelygin, Yuan Lin

GTC 2017

COOPERATIVE GROUPS VS BUILT-IN FUNCTIONS

Example: warp aggregated atomic

```
// increment the value at ptr by 1 and return the old value
__device__ int atomicAggInc(int *p);
```

```
coalesced_group g = coalesced_threads();


int res;
if (g.thread_rank() == 0)
   res = atomicAdd(p, g.size());
res = g.shfl(res, 0);
return g.thread_rank() + res;
```

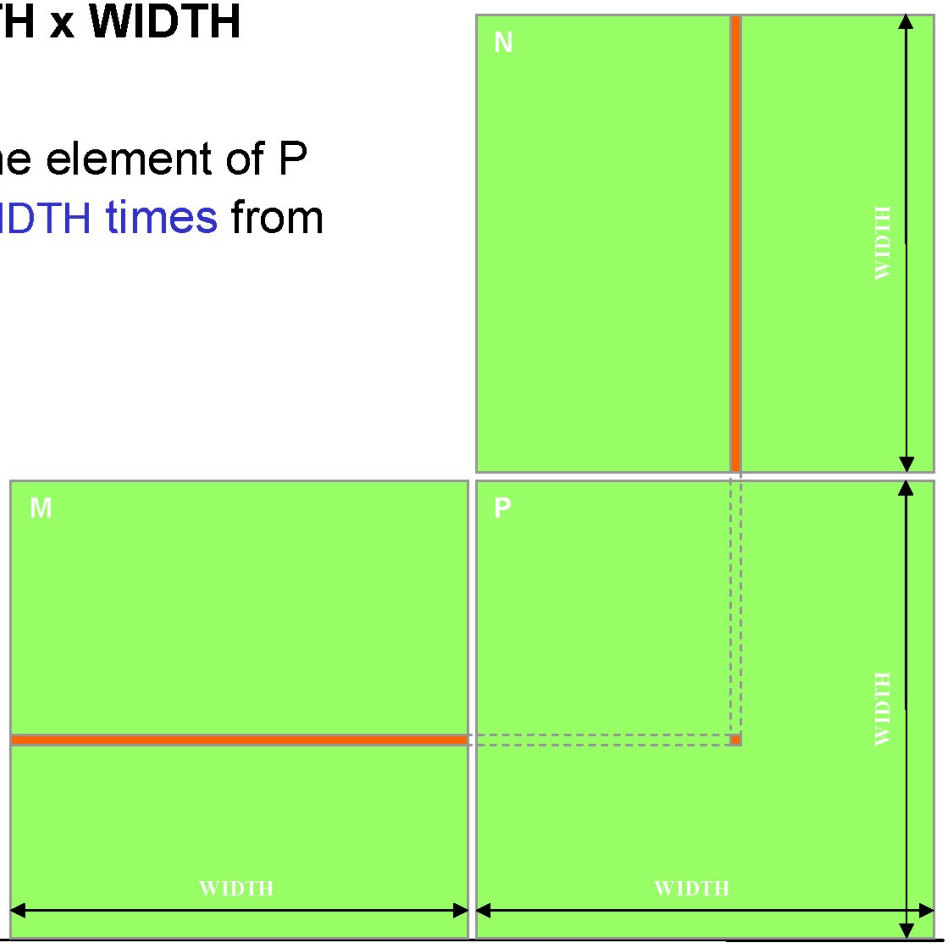
```
int mask = __activemask();
int rank = __popc(mask & __lanemask_lt());
int leader_lane = __ffs(mask) - 1;
int res;
if (rank == 0)
   res = atomicAdd(p, __popc(mask));
res = __shfl_sync(mask, res, leader_lane);
return rank + res;
```

Matrix-Matrix Multiplication

P=MN

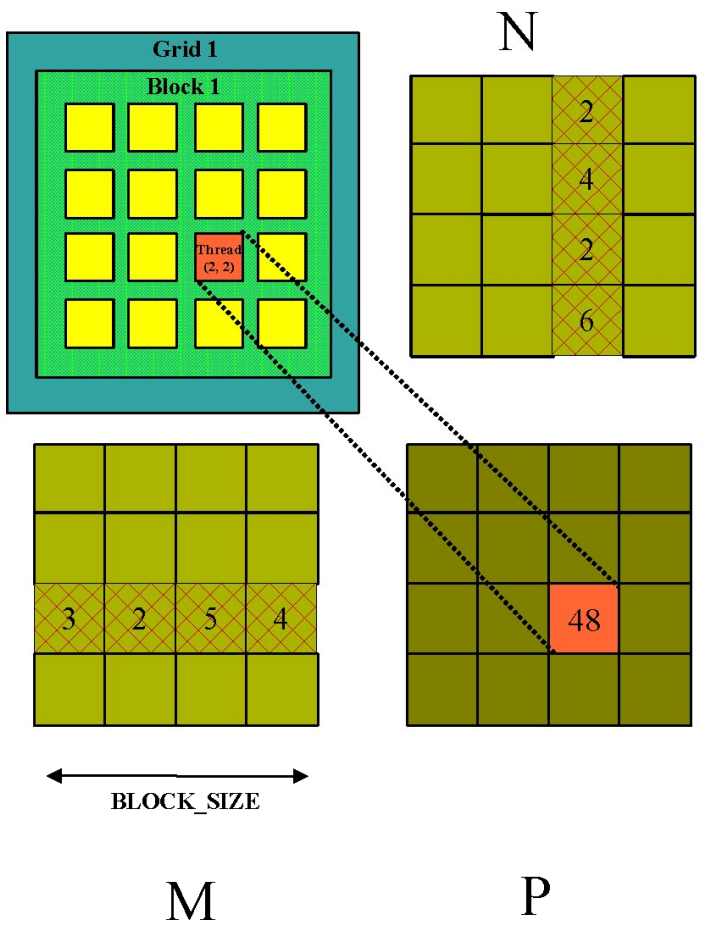
Programming Model: Square Matrix Multiplication

- **P = M * N of size WIDTH x WIDTH**
- **Without tiling:**
  - One thread handles one element of P
  - M and N are loaded WIDTH times from global memory



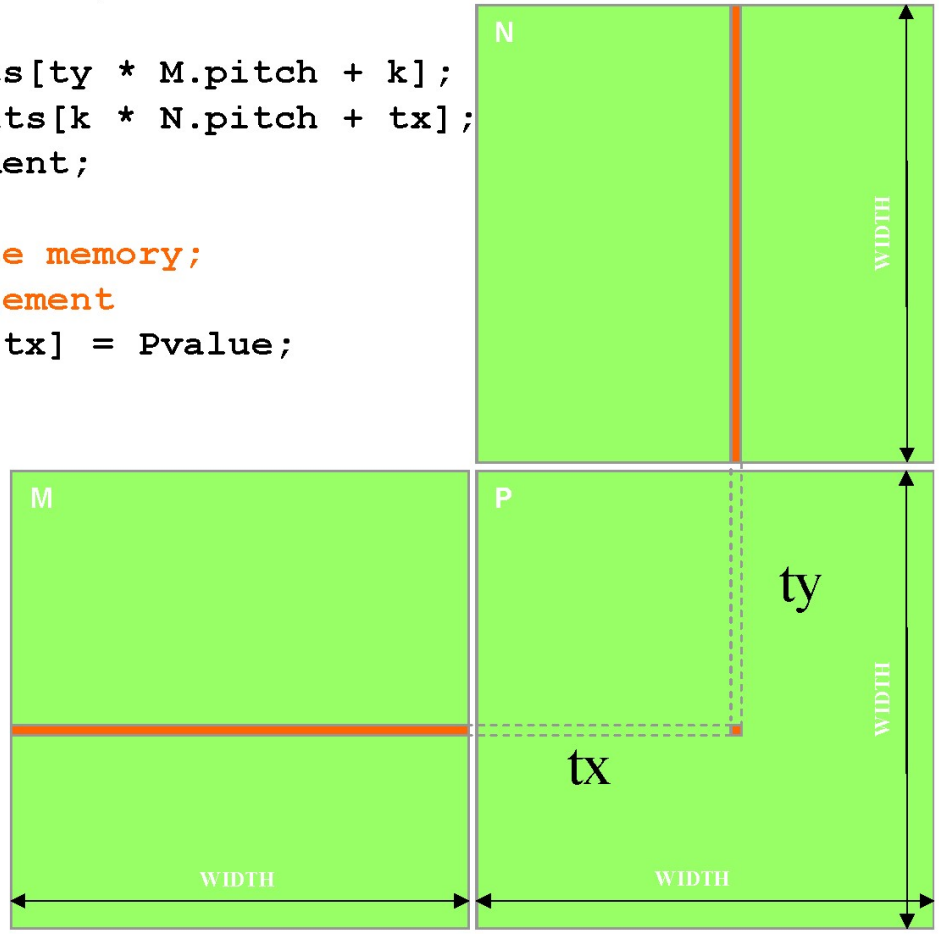
Multiply Using One Thread Block

- **One block of threads computes matrix P**
  - Each thread computes one element of P
- **Each thread**
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- **Size of matrix limited by the number of threads allowed in a thread block**



Matrix Multiplication
Device-Side Kernel Function (cont.)

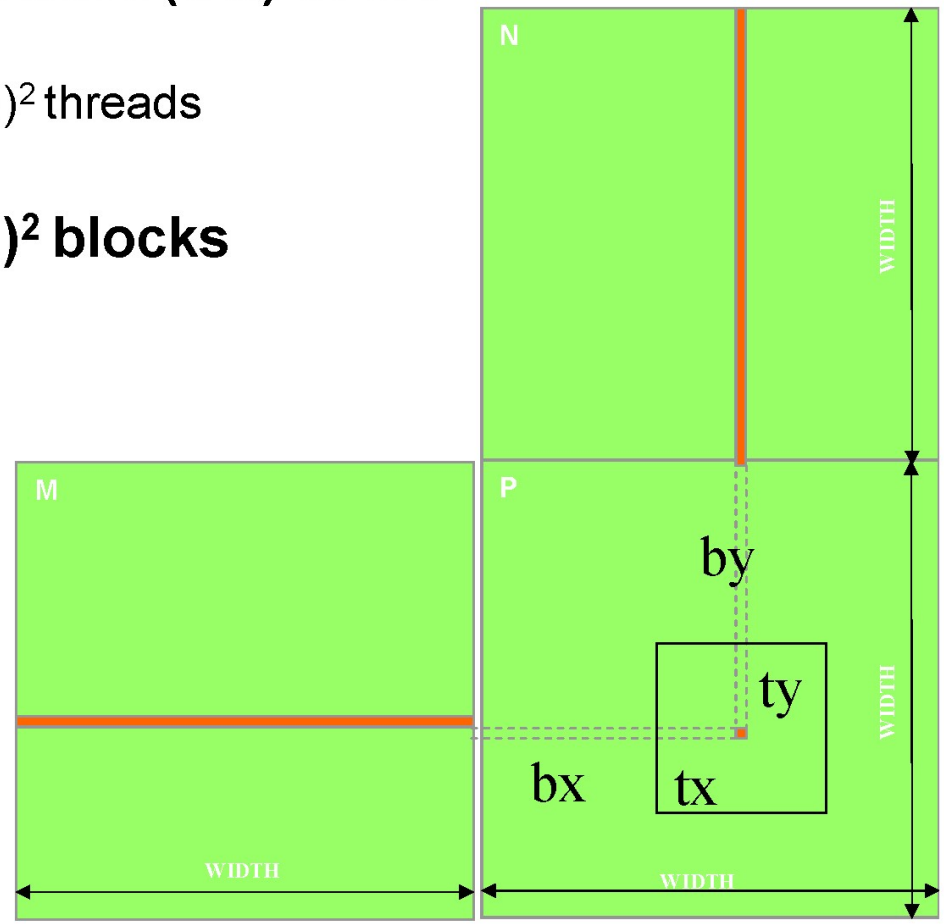
```
...
    for (int k = 0; k < M.width; ++k)
    {
      float Melement = M.elements[ty * M.pitch + k];
      float Nelement = Nd.elements[k * N.pitch + tx];
      Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty * blockDim.x+ tx] = Pvalue;
}
```



Handling Arbitrary Sized Square Matrices

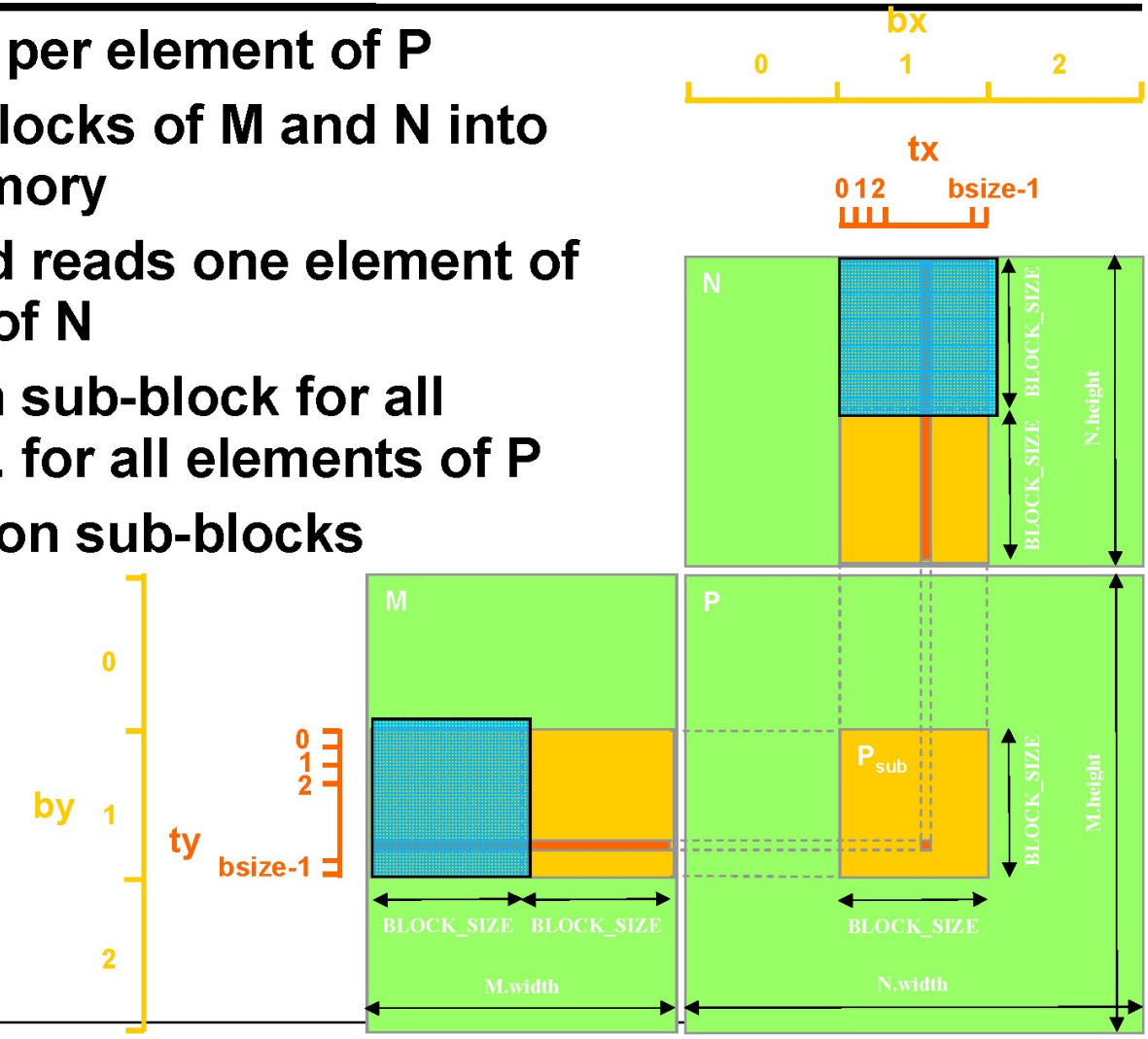
- **Have each 2D thread block to compute a $(BLOCK\_WIDTH)^2$ sub-matrix (tile) of the result matrix**
  - Each has $(BLOCK\_WIDTH)^2$ threads
- **Generate a 2D Grid of $(WIDTH/BLOCK\_WIDTH)^2$ blocks**

You still need to put a loop around the kernel call for cases where WIDTH is greater than Max grid size!



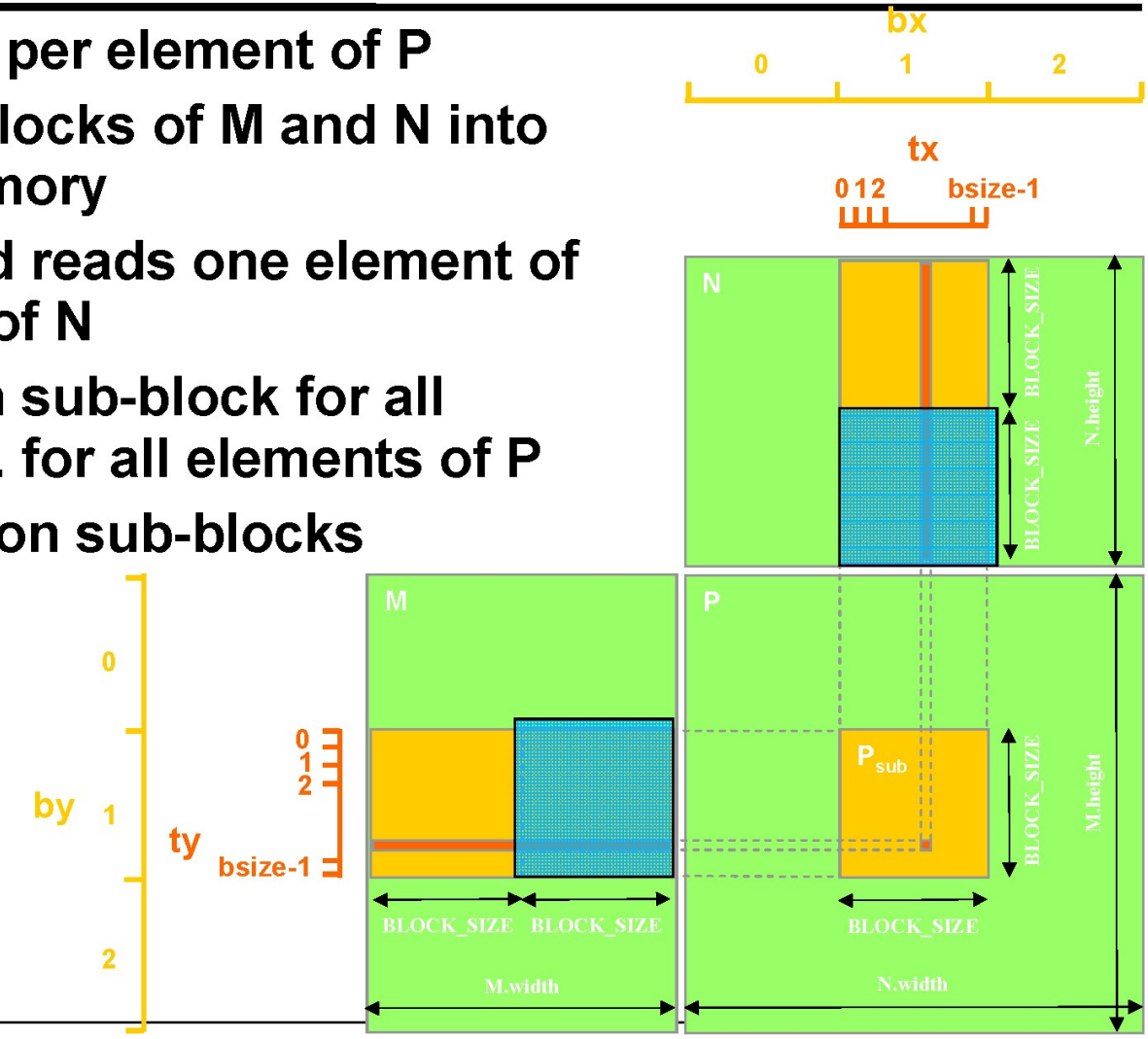
Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



Example: Matrix Multiplication (1)

- Copy matrices to device; invoke kernel; copy result matrix back to host

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void**)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc((void**)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);
```

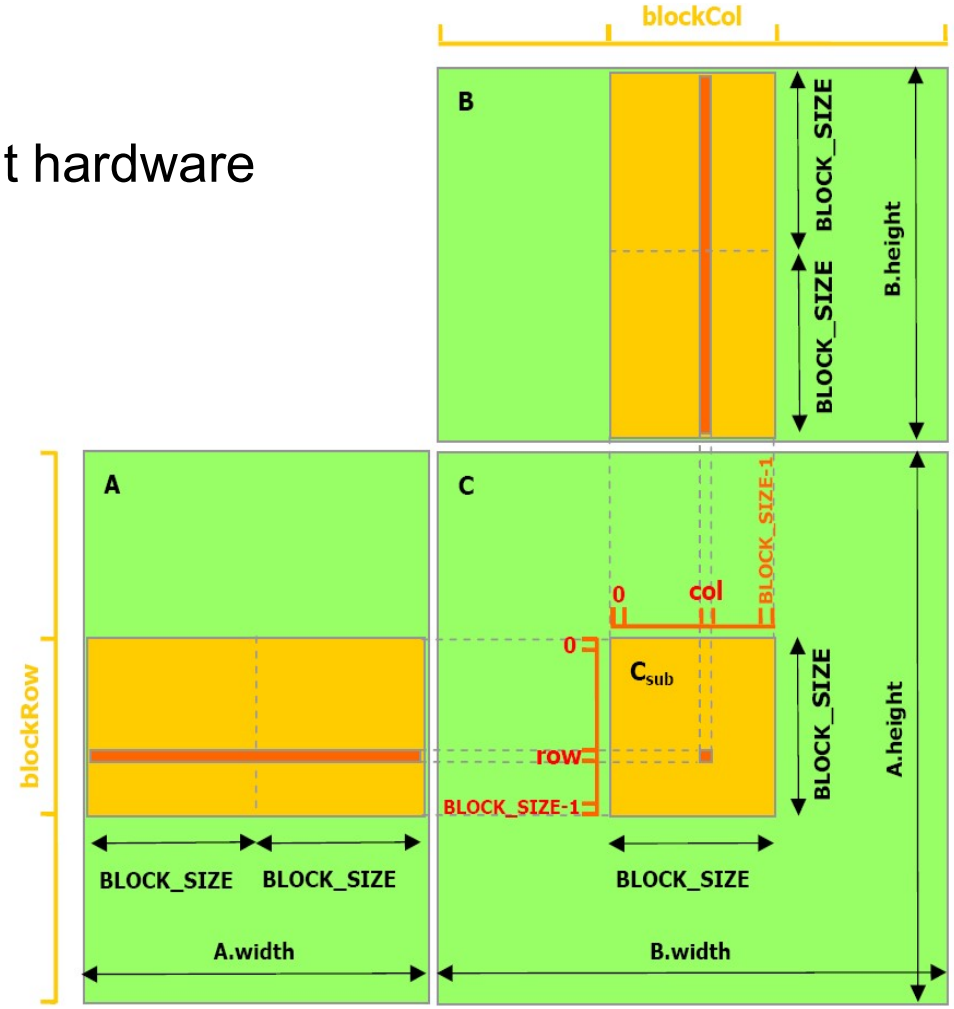
Example: Matrix Multiplication (2)

```
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc((void**)&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

Example: Matrix Multiplication (3)

- Multiply matrix block-wise
- Set BLOCK_SIZE for efficient hardware use, e.g., to 16 on cc. 1.x or 16 or 32 on cc. 2.x +

- Maximize parallelism
  - Launch as many threads per block as block elements
  - Each thread fetches one element of block
  - Perform row * column dot products in parallel

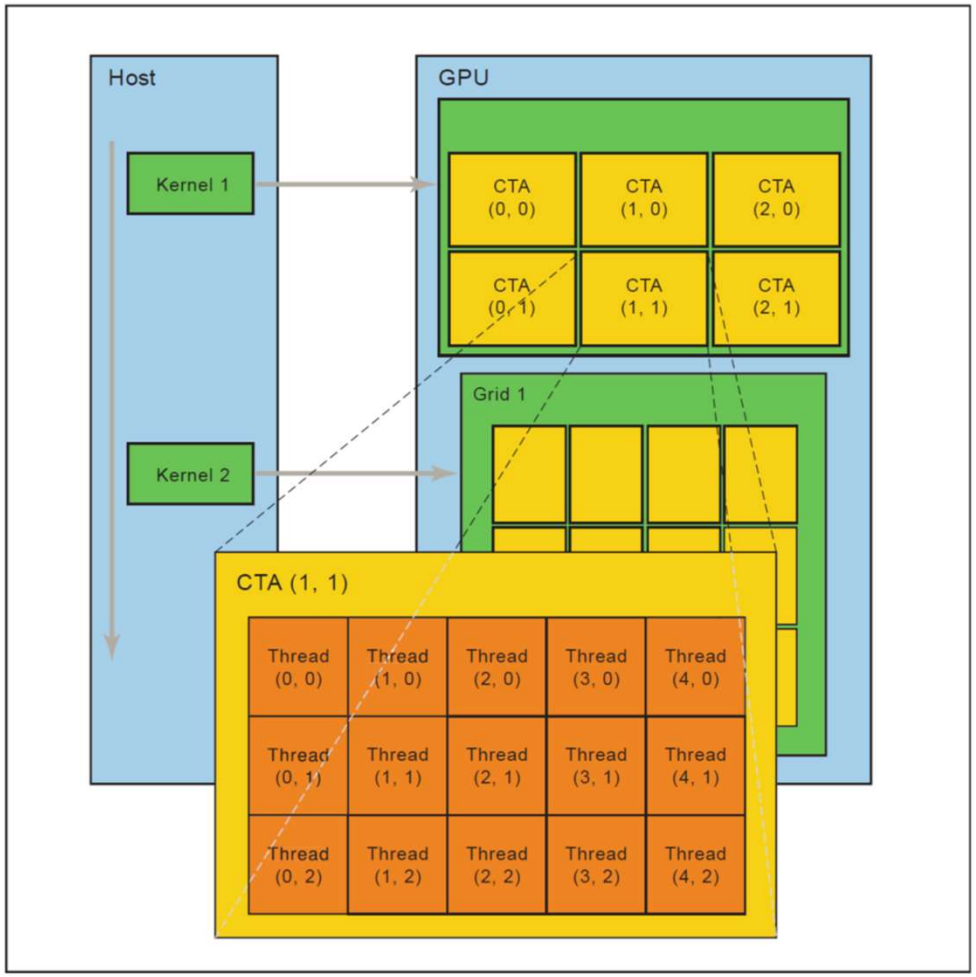


Example: Matrix Multiplication (4)

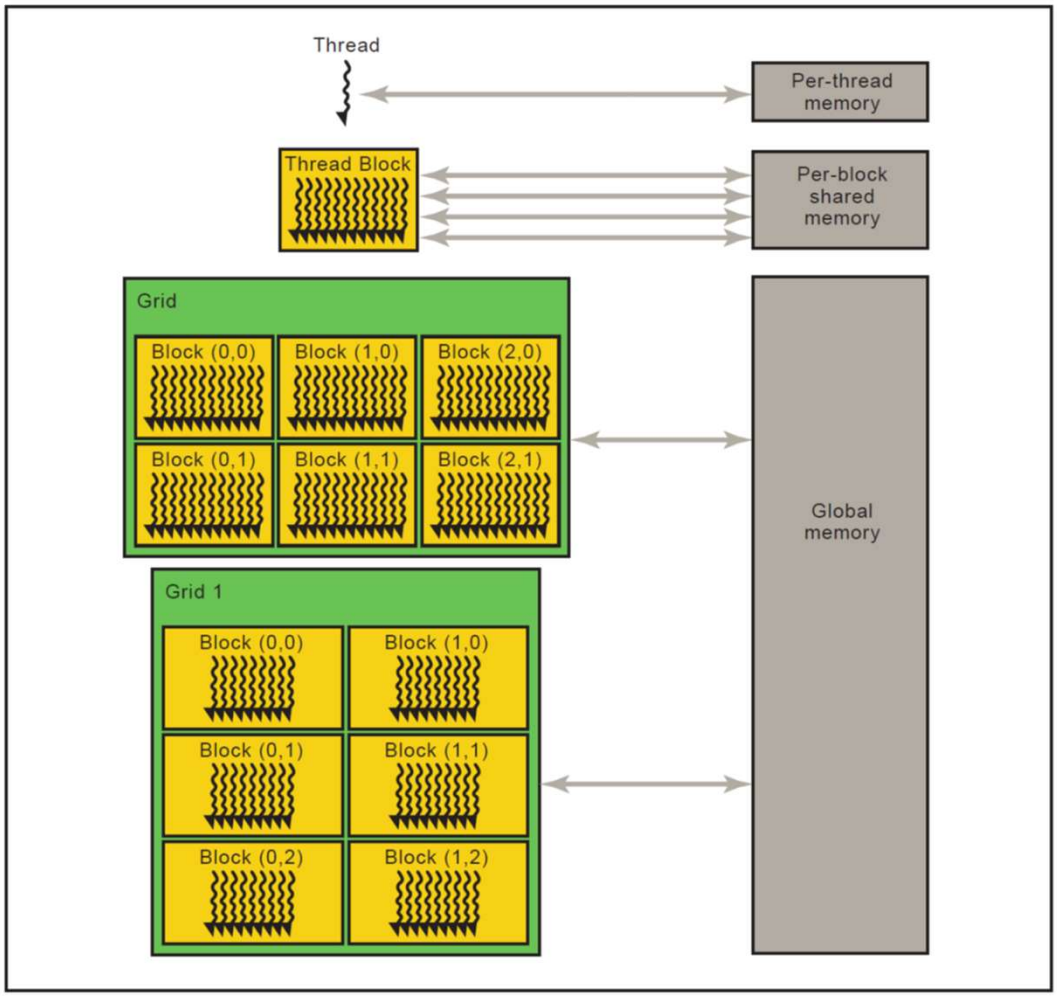
```
__global__ void MatrixMul( float *matA, float *matB, float *matC, int w )
{
      __shared__ float blockA[ BLOCK_SIZE ][ BLOCK_SIZE ];
      __shared__ float blockB[ BLOCK_SIZE ][ BLOCK_SIZE ];

      int bx = blockIdx.x; int tx = threadIdx.x;
      int by = blockIdx.y; int ty = threadIdx.y;

      int col = bx * BLOCK_SIZE + tx;
      int row = by * BLOCK_SIZE + ty;

      float out = 0.0f;
      for ( int m = 0; m < w / BLOCK_SIZE; m++ ) {

          blockA[ ty ][ tx ] = matA[ row * w +   m * BLOCK_SIZE + tx       ];
          blockB[ ty ][ tx ] = matB[ col     + ( m * BLOCK_SIZE + ty ) * w ];
          __syncthreads();

          for ( int k = 0; k < BLOCK_SIZE; k++ ) {
              out += blockA[ ty ][ k ] * blockB[ k ][ tx ];
          }
          __syncthreads();
      }

      matC[ row * w + col ] = out;
}
```

Caveat: for brevity, this code assumes matrix sizes are a multiple of the block size (either because they really are, or because padding is used; otherwise guard code would need to be added)

PTX Virtual Machine Model



PTX Virtual Machine Model



PTX Virtual Machine Model

this is a complete list of all PTX 7.1 instruction keywords

however, note that ultimately operand types, e.g., int vs. float, will result in different machine (SASS) instructions.

abs	div	or	sin	vavrg2, vavrg4
add	ex2	pmevent	slct	vmad
addc	exit	popc	sqrt	vmax
and	fma	prefetch	st	vmax2, vmax4
atom	isspacep	prefetchu	sub	vmin
bar	ld	prmt	subc	vmin2, vmin4
bfe	ldu	rcp	suld	vote
bfi	lg2	red	suq	vset
bfind	mad	rem	sured	vset2, vset4
bra	mad24	ret	sust	vshl
brev	madc	rsqrt	testp	vshr
brkpt	max	sad	tex	vsub
call	membar	selp	tld4	vsub2, vsub4
clz	min	set	trap	xor
cnot	mov	setp	txq	
copysign	mul	shf	vabsdiff	
cos	mul 24	shfl	vabsdiff2, vabsdiff4	
cvt	neg	shl	vadd	
cvta	not	shr	vadd2, vadd4	

PTX Code and Inline Assembly

```
        .reg     .b32 r1, r2;
        .global  .f32  array[N];

start:  mov.b32   r1, %tid.x;
        shl.b32   r1, r1, 2;          // shift thread id by 2 bits
        ld.global.b32 r2, array[r1];  // thread[tid] gets array[tid]
        add.f32   r2, r2, 0.5;        // add 1/2
```

```
__device__ int cube (int x)
{
  int y;
  asm("{\n\t"                          // use braces for local scope
      " reg .u32 t1;\n\t"              // temp reg t1,
      " mul.lo.u32 t1, %1, %1;\n\t"    // t1 = x * x
      " mul.lo.u32 %0, t1, %1;\n\t"    // y = t1 * x
      "}"
      : "=r"(y) : "r" (x));
  return y;
}
```

PTX (Parallel Thread Execution) Code

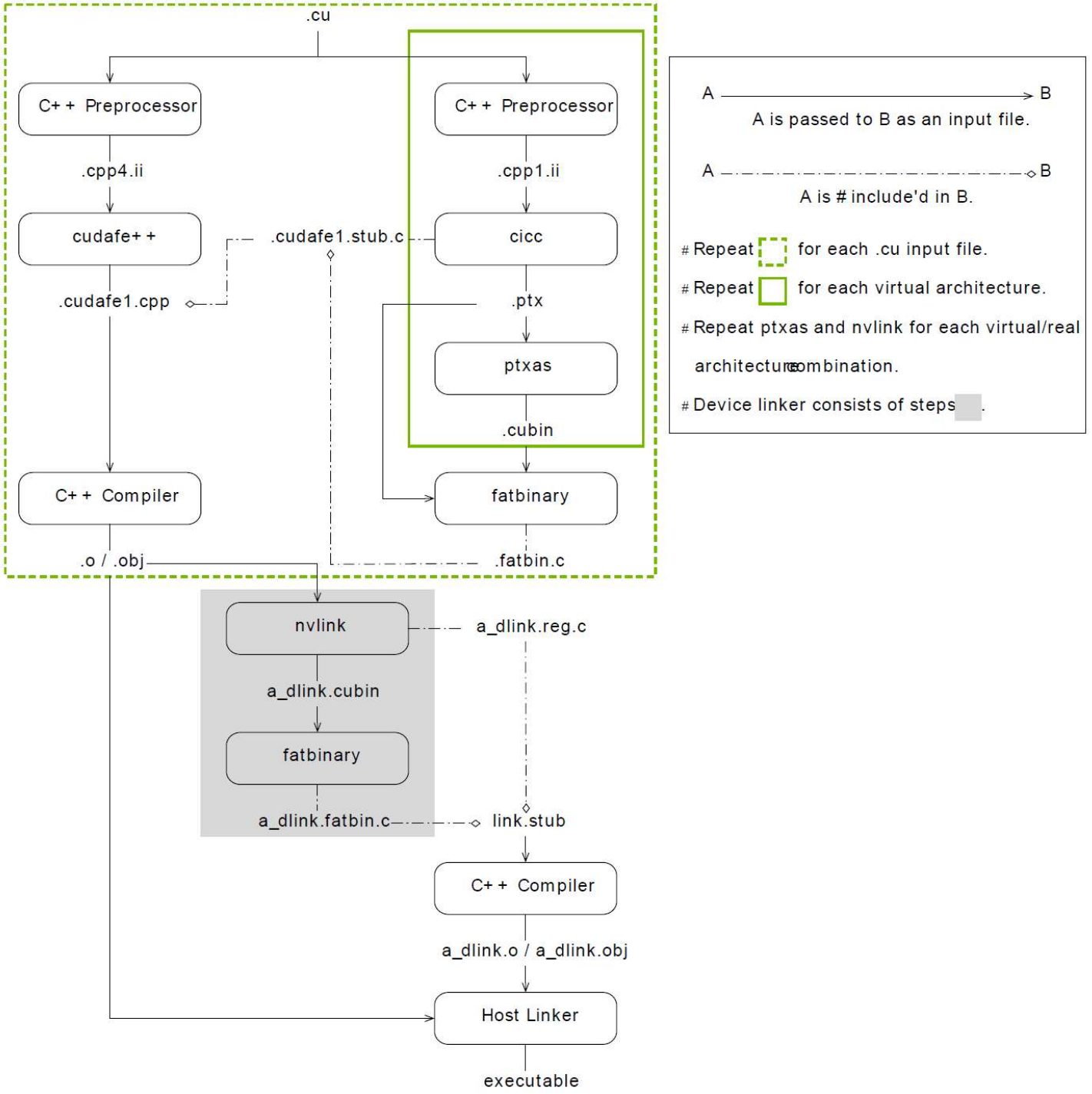
```
Fatbin ptx code:
================
arch = sm_20
code version = [4,0]
producer = cuda
host = linux
compile_size = 64bit
compressed
identifier = add.cu

.version 4.0
.target sm_20
.address_size 64

.visible .entry _Z3addPiS_S_(
.param .u64 _Z3addPiS_S_param_0,
.param .u64 _Z3addPiS_S_param_1,
.param .u64 _Z3addPiS_S_param_2
)
{
.reg .s32 %r<4>;
.reg .s64 %rd<7>;

ld.param.u64 %rd1, [_Z3addPiS_S_param_0];
ld.param.u64 %rd2, [_Z3addPiS_S_param_1];
ld.param.u64 %rd3, [_Z3addPiS_S_param_2];
cvta.to.global.u64 %rd4, %rd3;
cvta.to.global.u64 %rd5, %rd2;
cvta.to.global.u64 %rd6, %rd1;
ldu.global.u32 %r1, [%rd6];
ldu.global.u32 %r2, [%rd5];
add.s32 %r3, %r2, %r1;
st.global.u32 [%rd4], %r3;
ret;
}
```

SASS (Streaming Assembler) Code

```
$ cuobjdump a.out -ptx -sass
Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit
identifier = add.cu

        code for sm_20
                Function : _Z3addPiS_S_
        .headerflags    @"EF_CUDA_SM20 EF_CUDA_PTX_SM(EF_CUDA_SM20)"
        /*0000*/          MOV R1, c[0x1][0x100];  /* 0x2800440400005de4 */
        /*0008*/          MOV R6, c[0x0][0x20];   /* 0x2800400080019de4 */
        /*0010*/          MOV R7, c[0x0][0x24];   /* 0x280040009001dde4 */
        /*0018*/          MOV R2, c[0x0][0x28];   /* 0x28004000a0009de4 */
        /*0020*/          MOV R3, c[0x0][0x2c];   /* 0x28004000b000dde4 */
        /*0028*/          LDU.E R0, [R6];         /* 0x8c00000000601c85 */
        /*0030*/          MOV R4, c[0x0][0x30];   /* 0x28004000c0011de4 */
        /*0038*/          LDU.E R2, [R2];         /* 0x8c00000000209c85 */
        /*0040*/          MOV R5, c[0x0][0x34];   /* 0x28004000d0015de4 */
        /*0048*/          IADD R0, R2, R0;        /* 0x4800000000201c03 */
        /*0050*/          ST.E [R4], R0;          /* 0x9400000000401c85 */
        /*0058*/          EXIT;                   /* 0x8000000000001de7 */
                ............................
```

NVCC

CUDA compilation trajectory

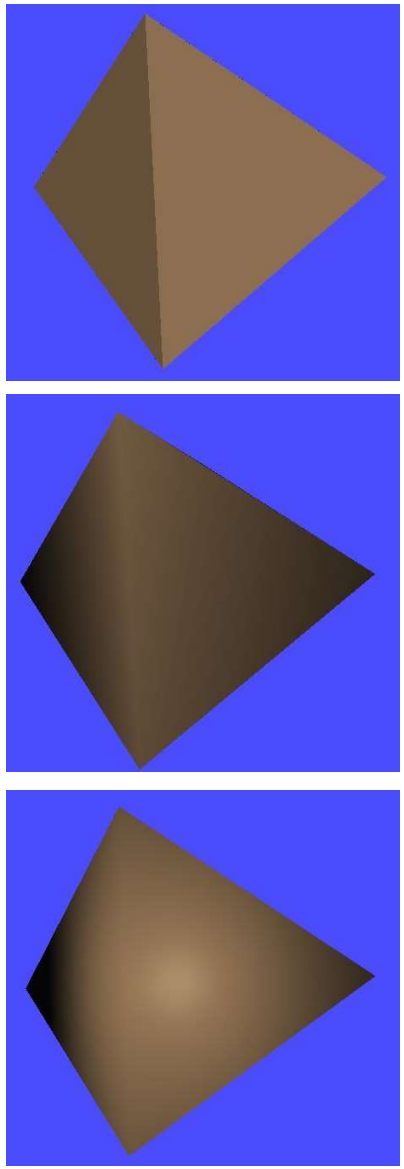


GPU Texturing

Rage / id Tech 5 (id Software)

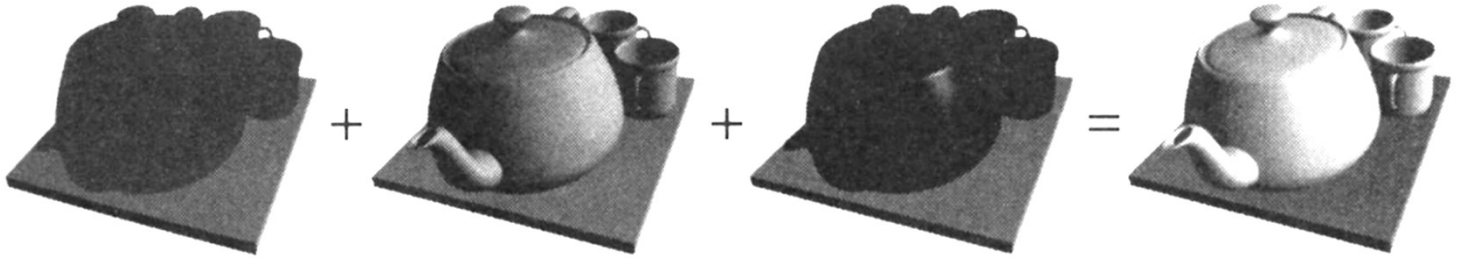
Remember: Basic Shading

- Flat shading
  - compute light interaction per polygon
  - the whole polygon has the same color
- Gouraud shading
  - compute light interaction per vertex
  - interpolate the colors
- Phong shading
  - interpolate normals per pixel
- Remember: difference between
  - Phong Lighting Model
  - Phong Shading



Traditional OpenGL Lighting

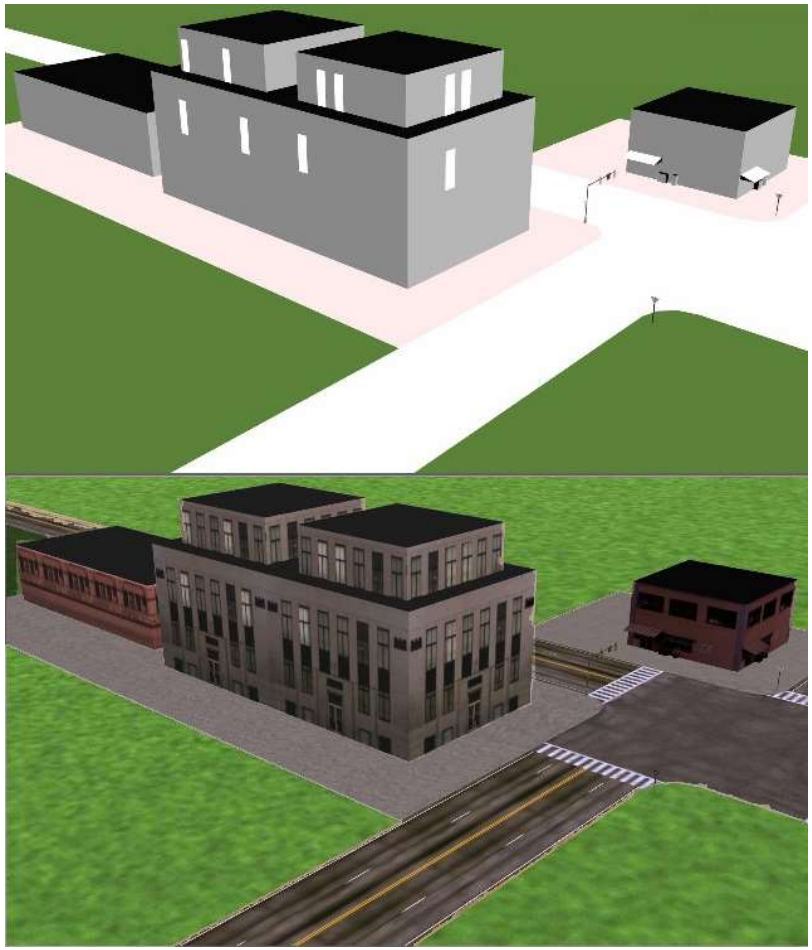
- Phong lighting model at each vertex (glLight, …)
- Local model only (no shadows, radiosity, …)
- ambient + diffuse + specular (glMaterial!)



- Fixed function: Gouraud shading
  - Note: need to interpolate specular separately!
- Phong shading: evaluate Phong lighting model in fragment shader (per-fragment evaluation!)

Why Texturing?

- Idea: enhance visual appearance of surfaces by applying fine / high-resolution details



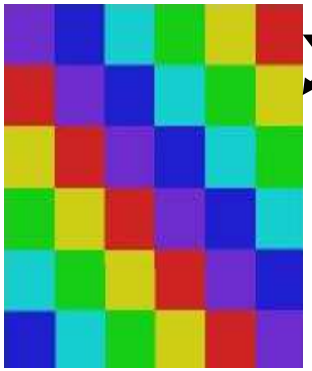
OpenGL Texture Mapping

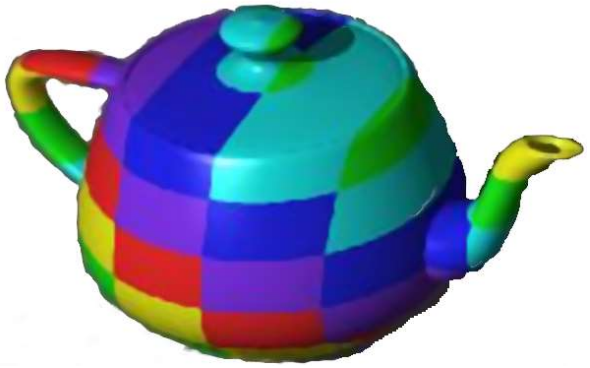
- Basis for most real-time rendering effects
- Look and feel of a surface
- Definition:
  - A *regularly sampled function* that is mapped onto every *fragment* of a surface
  - Traditionally an image, but…
- Can hold arbitrary information
  - Textures become general data structures
  - Sampled and interpreted by fragment programs
  - Can render into textures → important!

Types of Textures

- Spatial layout
  - Cartesian grids: 1D, 2D, 3D, 2D_ARRAY, …
  - Cube maps, …
- Formats (too many), e.g. OpenGL
  - GL_LUMINANCE16_ALPHA16
  - GL_RGB8, GL_RGBA8, …: integer texture formats
  - GL_RGB16F, GL_RGBA32F, …: float texture formats
  - compressed formats, high dynamic range formats, …
- External (CPU) format vs. internal (GPU) format
  - OpenGL driver converts from external to internal

Texturing: General Approach





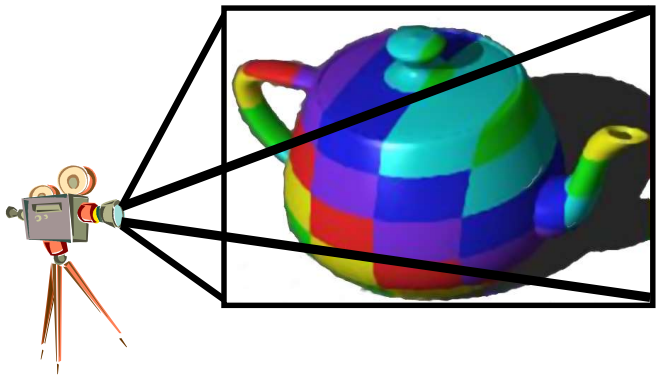
Texture space *(u,v)*

Object space (x_O, y_O, z_O)

Image Space (x_I, y_I)

Parametrization

Rendering (Projection etc.)



Texture Mapping

2D (3D) Texture Space

Texture Transformation

2D Object Parameters

Parameterization

3D Object Space

Model Transformation

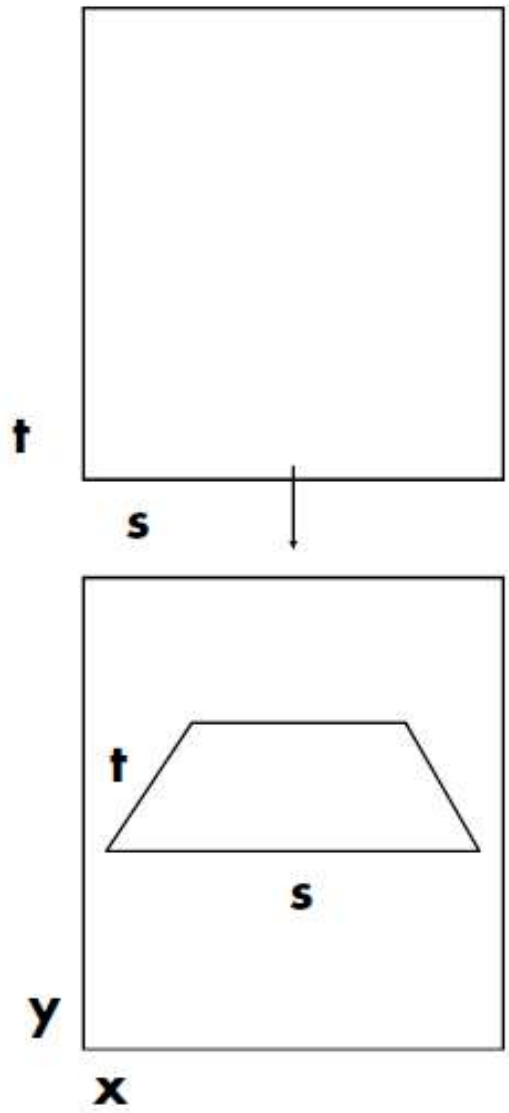
3D World Space

Viewing Transformation

3D Camera Space

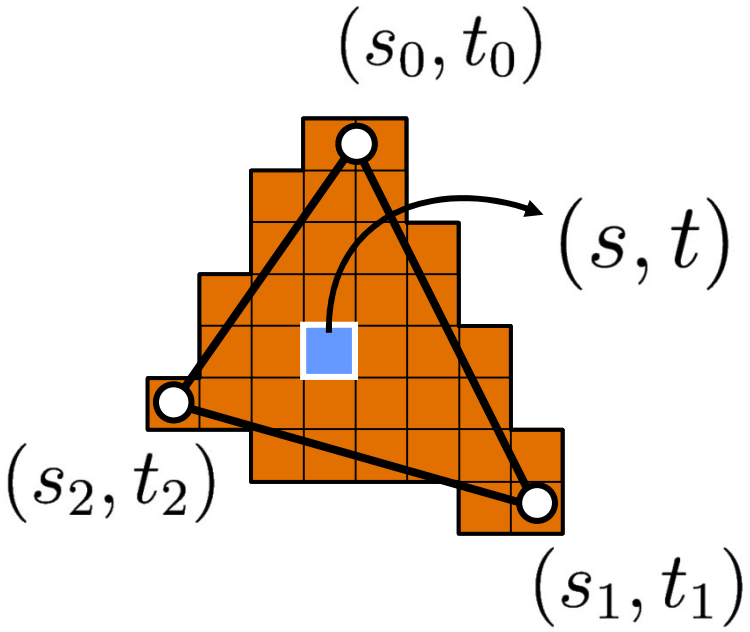
Projection

2D Image Space

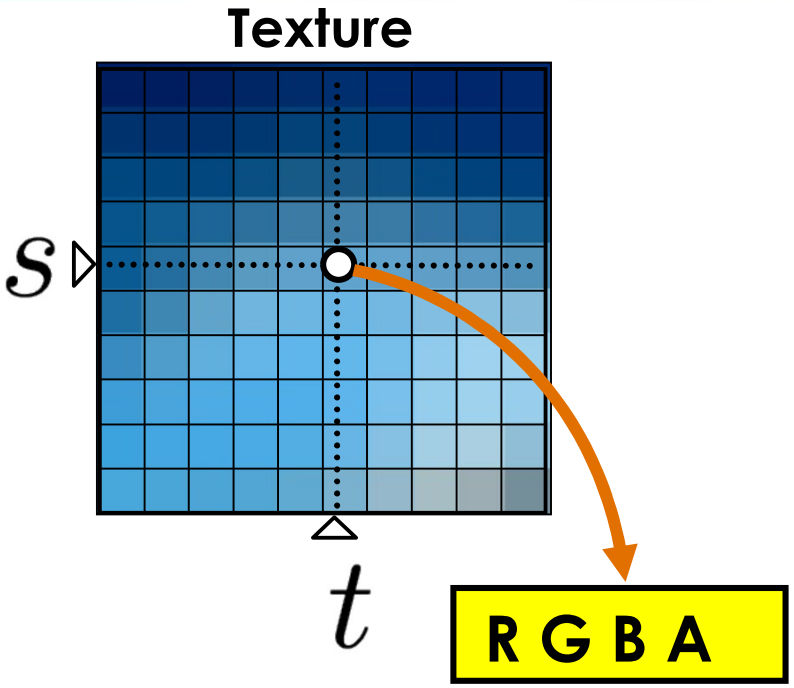


Kurt Akeley, Pat Hanrahan

2D Texture Mapping

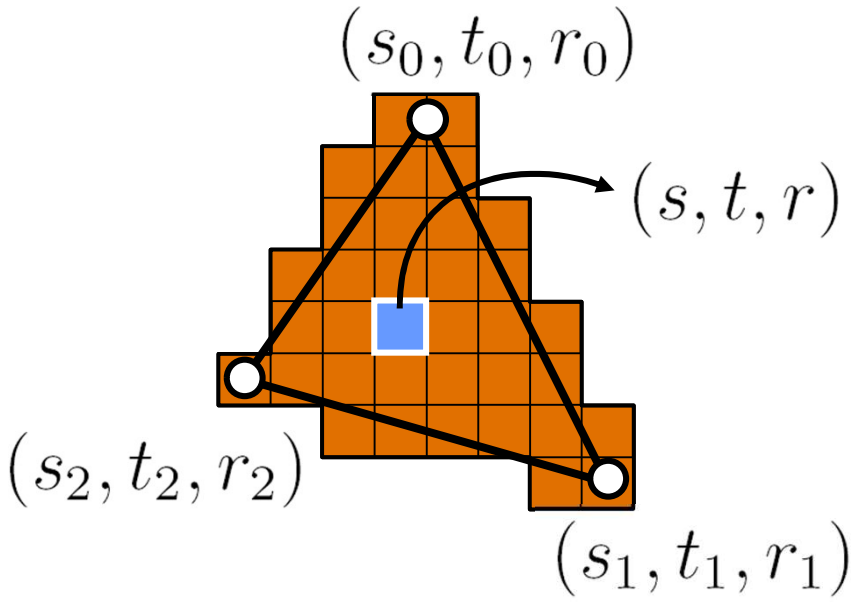


For each fragment:
interpolate the
texture coordinates
(barycentric)
Or:
Use arbitrary, computed coordinates

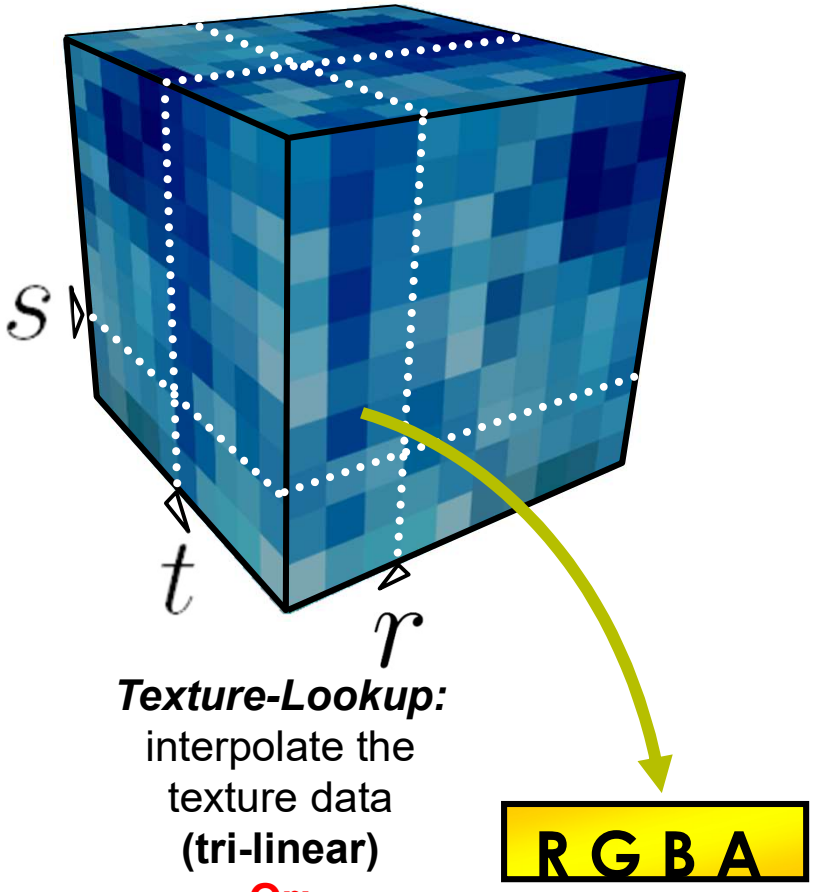


Texture-Lookup:
interpolate the
texture data
(bi-linear)
Or:
Nearest-neighbor for "array lookup"

3D Texture Mapping



For each fragment:
interpolate the
texture coordinates
(barycentric)
Or:
Use arbitrary, computed coordinates



Texture-Lookup:
interpolate the
texture data
(tri-linear)
Or:
Nearest-neighbor for "array lookup"

Thank you.