GPU TECHNOLOGY CONFERENCE

From Brook to CUDA

GPU Technology Conference

NVIDIA.

A 50 Second Tutorial on GPU Programming
by Ian Buck

Adding two vectors in C is pretty easy …

```
for (i=0; i<n; i++)
  c[i] = a[i] + b[i];
```

On the GPU, it's a wee bit more complicated …

First, you'll want to create a floating point PBuffer

Of course, there is
different code for
NVIDIA and ATI,
OpenGL and DirectX,
Windows, Linux, OS X

… naturally

You’ll want to create some floating point textures

Don’t forget to turn off filtering otherwise everything will run in software mode

good luck finding that in the documentation …

You’ll need to write the “add” shader…

```
char singleFetch[] =
"!!ARBfp1.0\n"
"TEMP R0;\n"
"TEMP R1;\n"
"TEX R0, fragment.texcoord[0], texture[0], RECT;\n"
"TEX R1, fragment.texcoord[0], texture[1], RECT;\n"
"ADD result.color, R0, R1;\n"
"END\n";
```

Copy the data to the GPU …

```
glTexSubImage2D(GL_TEXTURE_RECTANGLE_EXT, 0,
                0, 0, width, height, GLformat(ncomp[i]),
                GL_FLOAT, t);
```

Render a shaded quad…

```
glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB,
                                  pass_id[pass_idx]);

glBegin(GL_TRIANGLES);
glMultiTexCoord4fARB(GL_TEXTURE0_ARB,
               f1[i].x, f2[i].y, 0.0f, 1.0f);
glVertex2f(-1.0f, 3.0f);
glMultiTexCoord4fARB(GL_TEXTURE0_ARB,
               f1[i].x, f2[i].y, 0.0f, 1.0f);
glMultiTexCoord4fARB(GL_TEXTURE0_ARB+i,
               f1[i].x, f2[i].y, 0.0f, 1.0f);
glVertex2f(-1.0f, -1.0f);
glMultiTexCoord4fARB(GL_TEXTURE0_ARB+i,
               f1[i].x, f2[i].y, 0.0f, 1.0f);
glVertex2f(3.0f, -1.0f);
glEnd();
CHECK_GL();
```

Boy…

that sucked.

Read back from the GPU …

```
glReadPixels (0, 0, width, height, GLformat(ncomp[i]),
              GL_FLOAT, t);
```

Congratulations, you’ve successfully added two vectors

History....

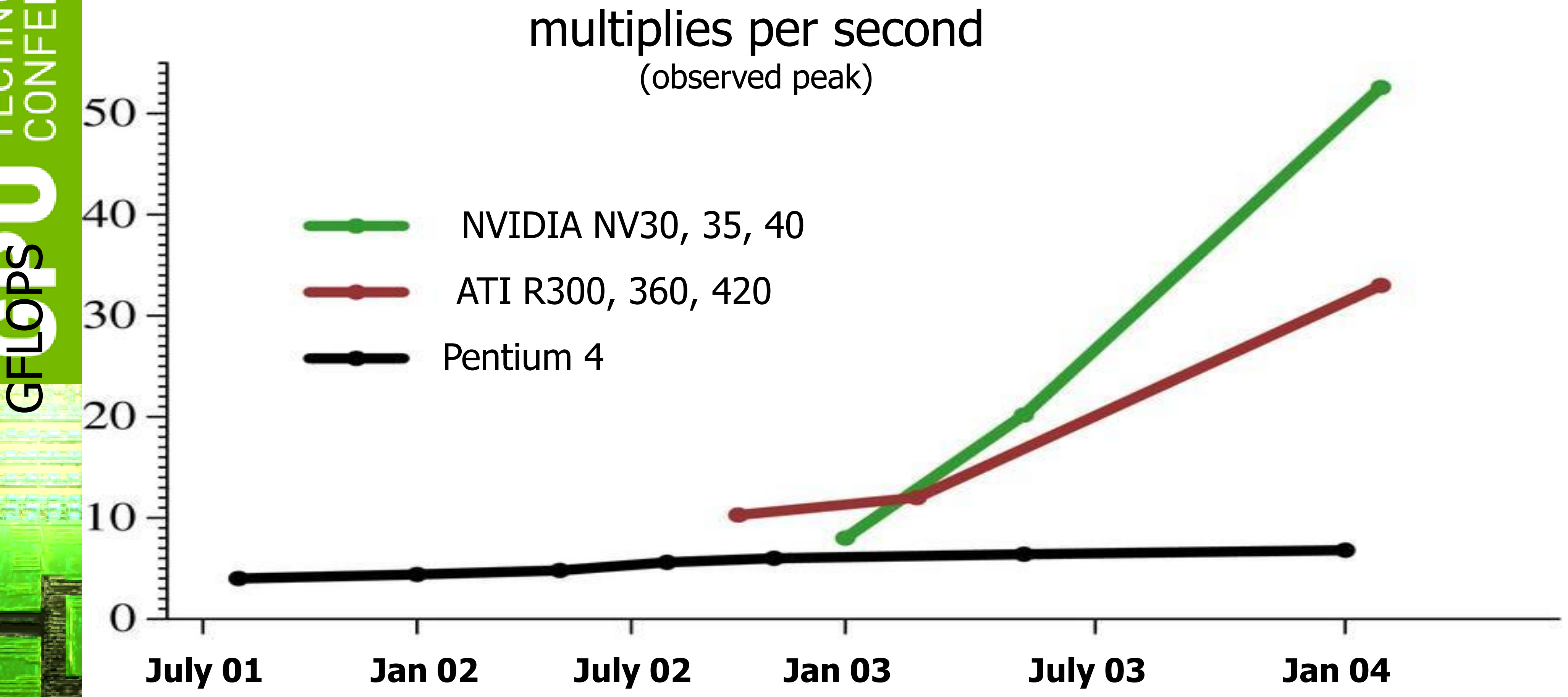
Stream Computing on Graphics Hardware

Ian Buck

Special University Oral Examination
Computer Science Department
Stanford University
December 17, 2004

GPGPU in 2004

recent trends



GPU history

NVIDIA historicals

	Product	Process	**Trans**	**MHz**	**GFLOPS (MUL)**
Aug-02	GeForce FX5800	0.13	**121M**	**500**	**8**
Jan-03	GeForce FX5900	0.13	**130M**	**475**	**20**
Dec-03	GeForce 6800	0.13	**222M**	**400**	**53**

translating transistors into performance

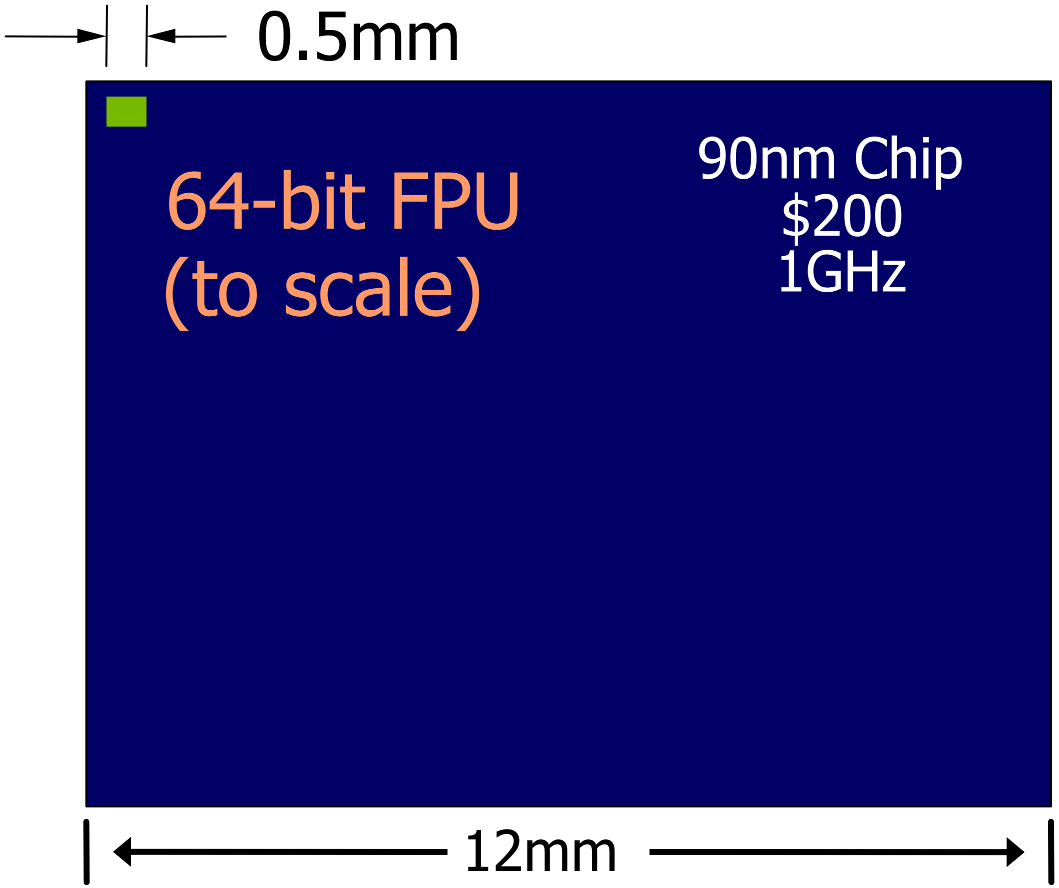
– 1.8x increase of transistors

– 20% *decrease* in clock rate

– 6.6x GFLOP speedup

compute is cheap

- **parallelism**
  - to keep 100s of ALUs per chip busy
- **shading is highly parallel**
  - millions of fragments per frame

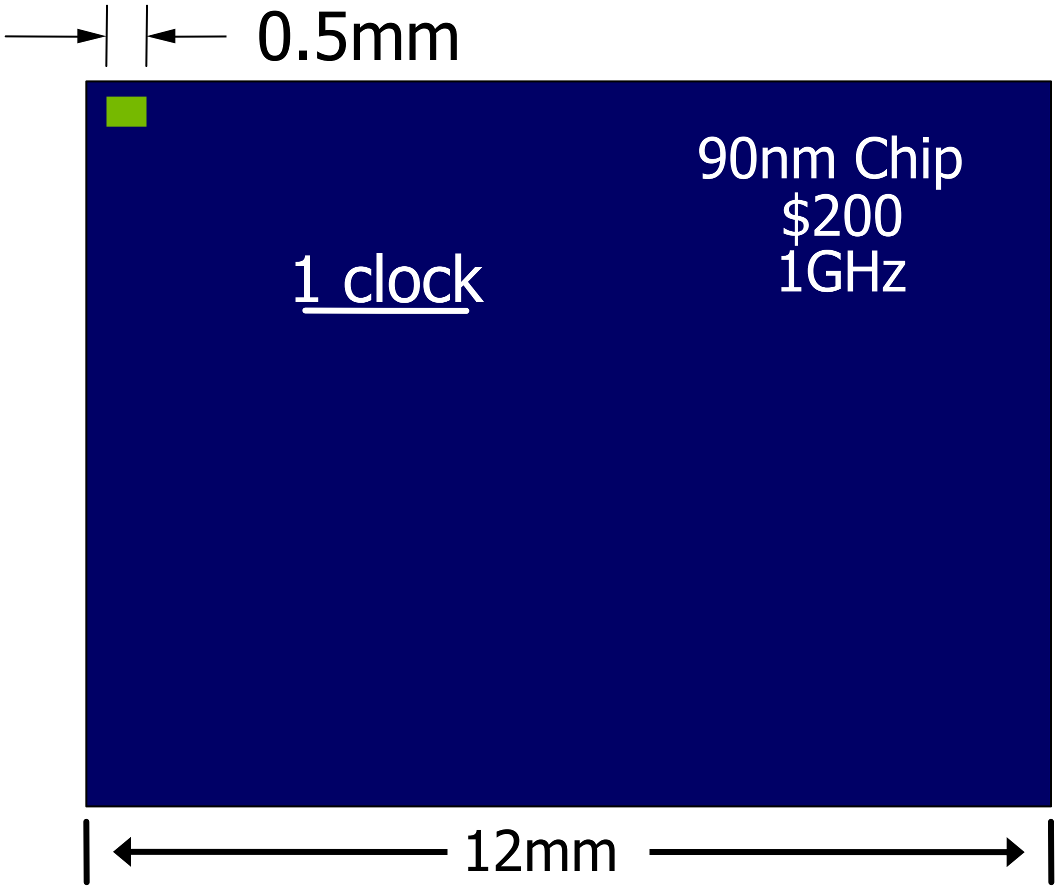


NVIDIA.

courtesy of Bill Dally

…but bandwidth is expensive

- **latency tolerance**
  - to cover 500 cycle remote memory access time
- **locality**
  - to match 20Tb/s ALU bandwidth to ~100Gb/s chip bandwidth



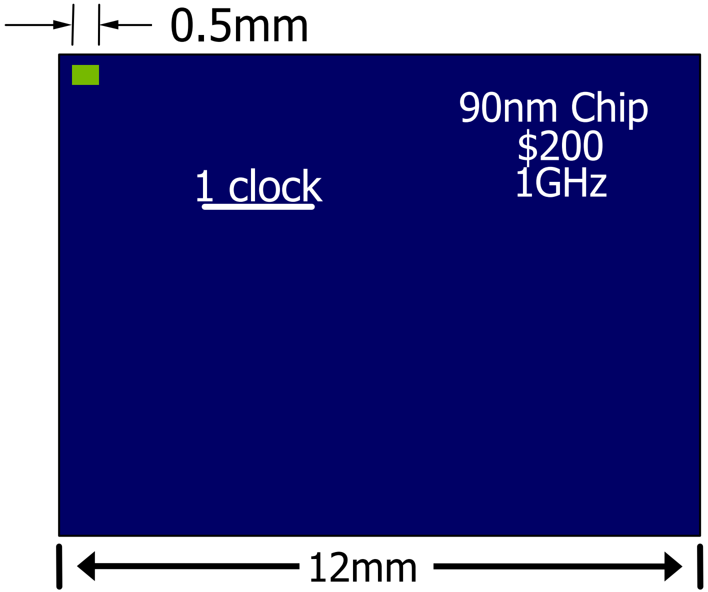
NVIDIA.

courtesy of Bill Dally

arithmetic intensity

- **shading is compute intensive**
  - 100s of floating point operations
  - output 1 32-bit color value

- **arithmetic intensity**
  - compute to bandwidth ratio



can we structure our computation in a similar way?

Brook language

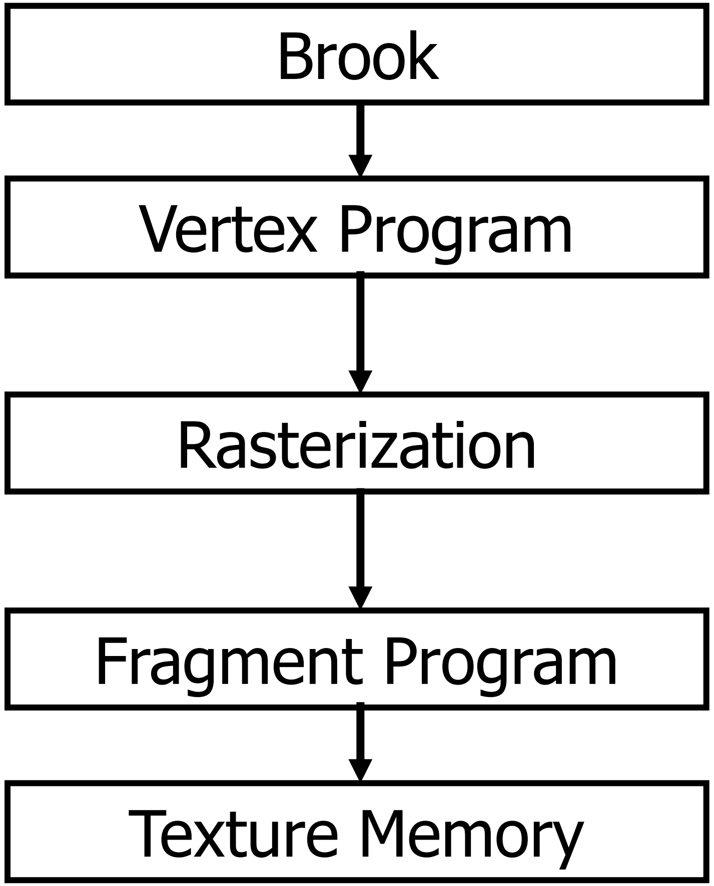
C with streams

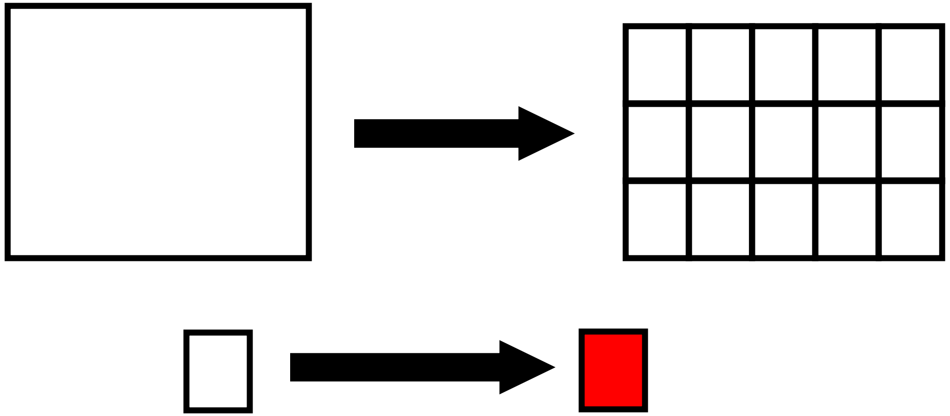
- streams
  - collection of records requiring similar computation
    - particle positions, voxels, FEM cell, …

```
Ray r<200>;
float3 velocityfield<100,100,100>;
```

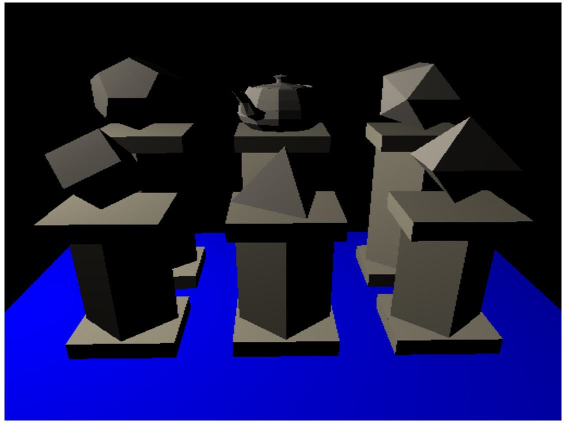
  - similar to arrays, but…
    - index operations disallowed: `position[i]`
    - read/write stream operators:

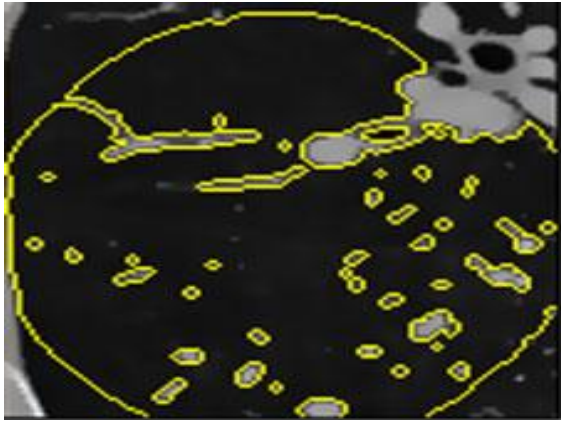
issuing compute geometry



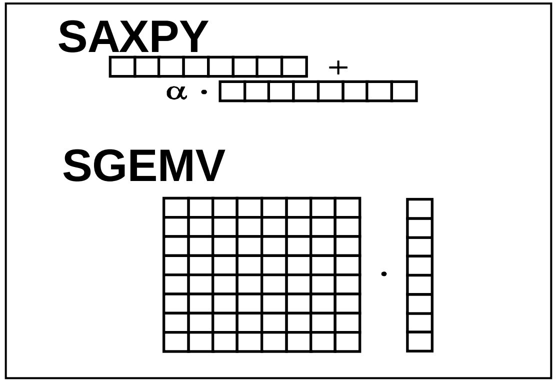


Brook Applications


ray-tracer


segmentation


fft edge detect



linear algebra

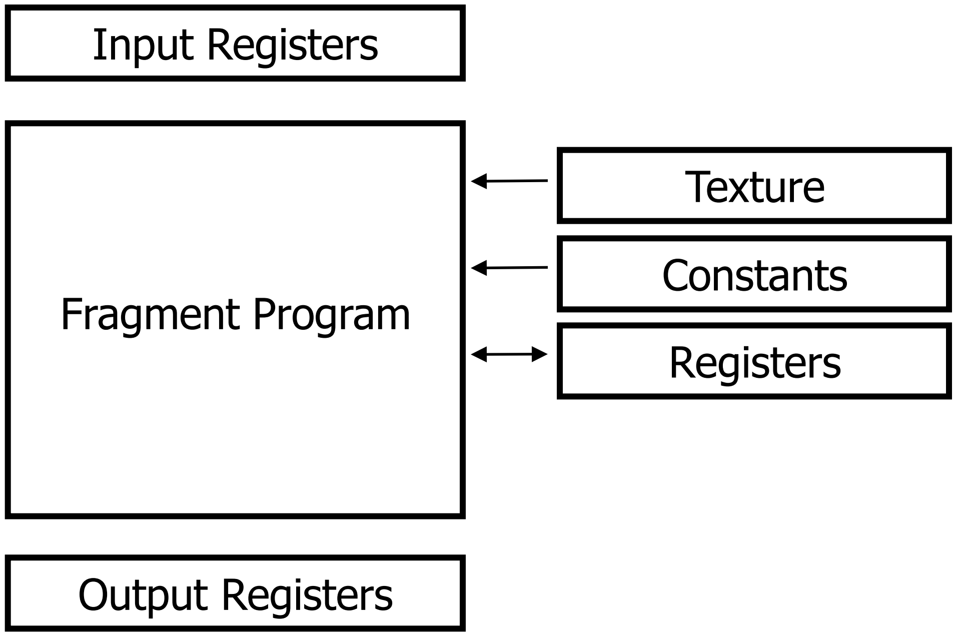
Legacy GPGPU

- Brook was great but…
  - Lived within the constraints of graphics
    - Constrained streaming programming model

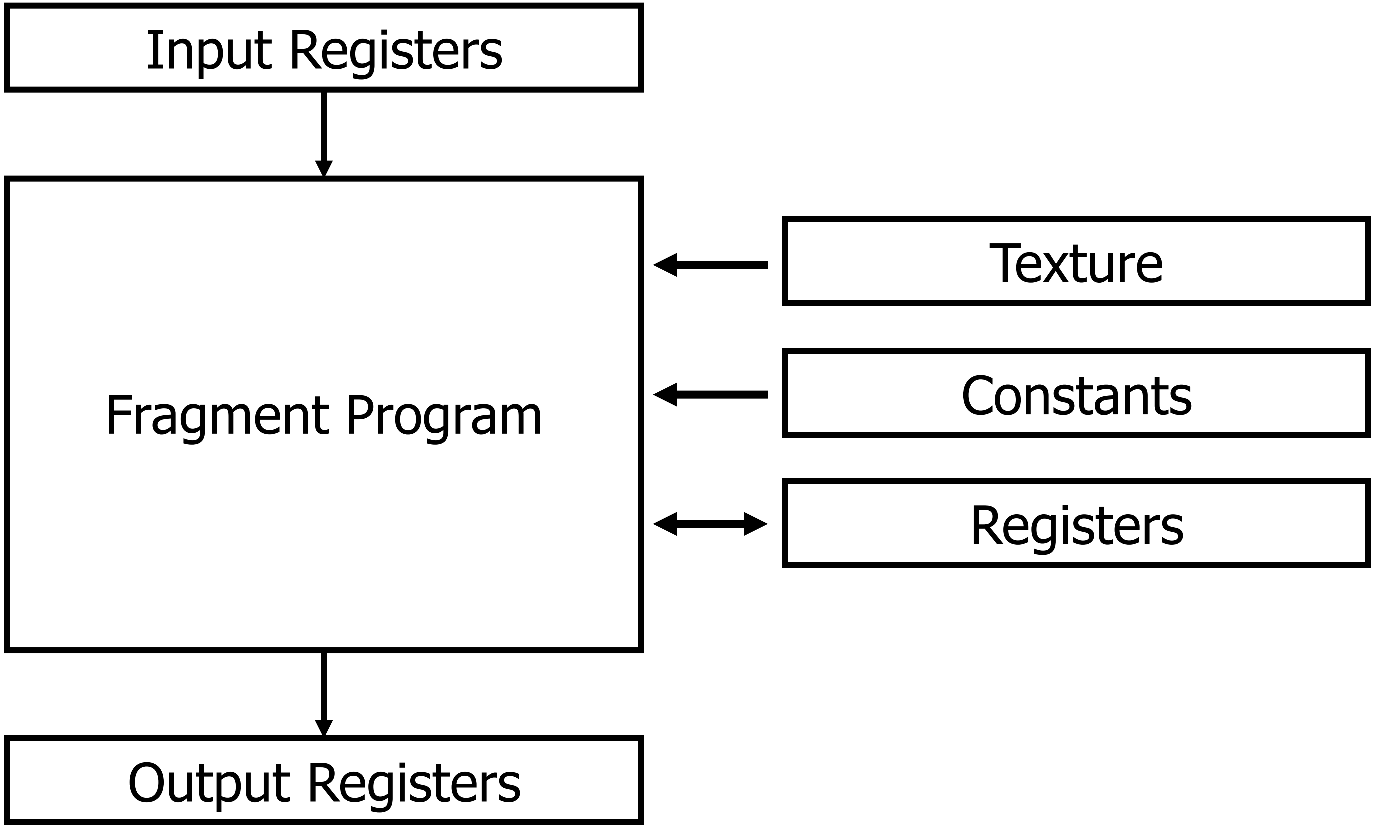
- How can we improve GPUs to be better computing platforms?

Challenges

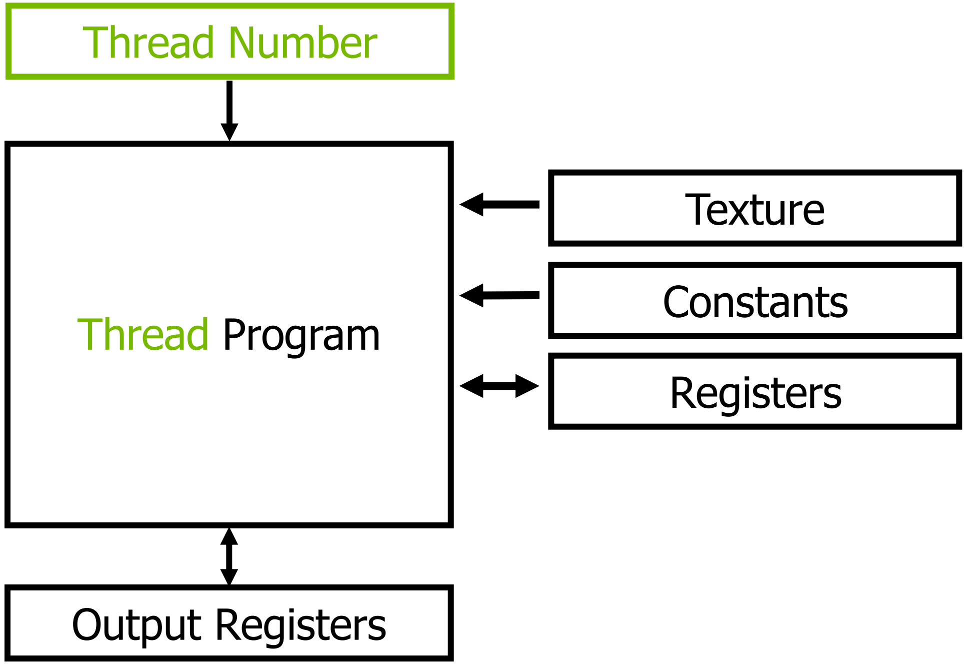
- Graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
- Instruction sets
  - Integer & bit ops
- Communication limited
  - Between pixels
  - Scatter a[i] = p



GeForce 7800 Pixel



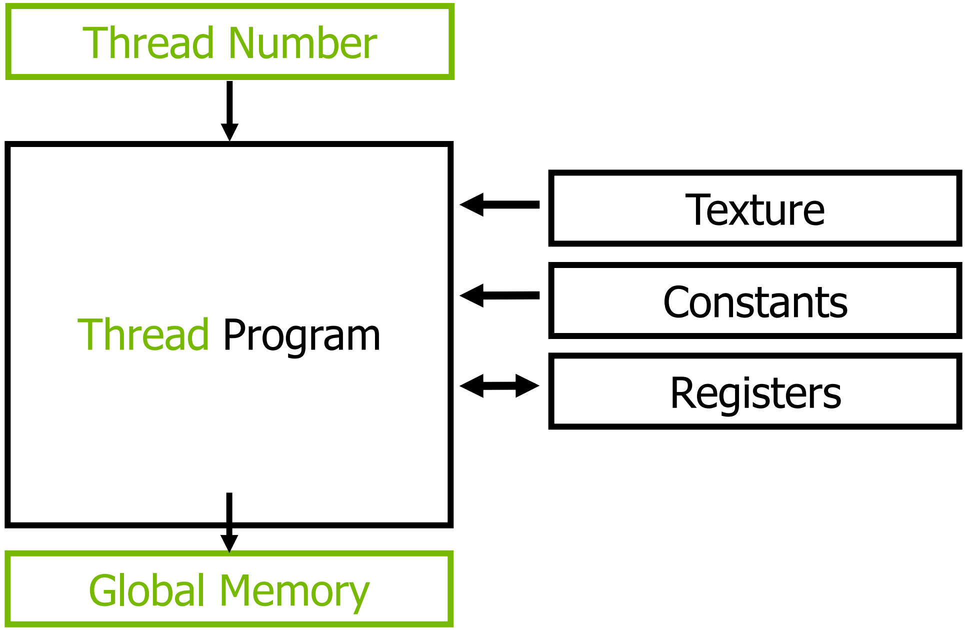
Thread Programs



Features

- Millions of instructions
- Full Integer and Bit instructions
- No limits on branching, looping
- 1D, 2D, or 3D thread ID allocation

Global Memory

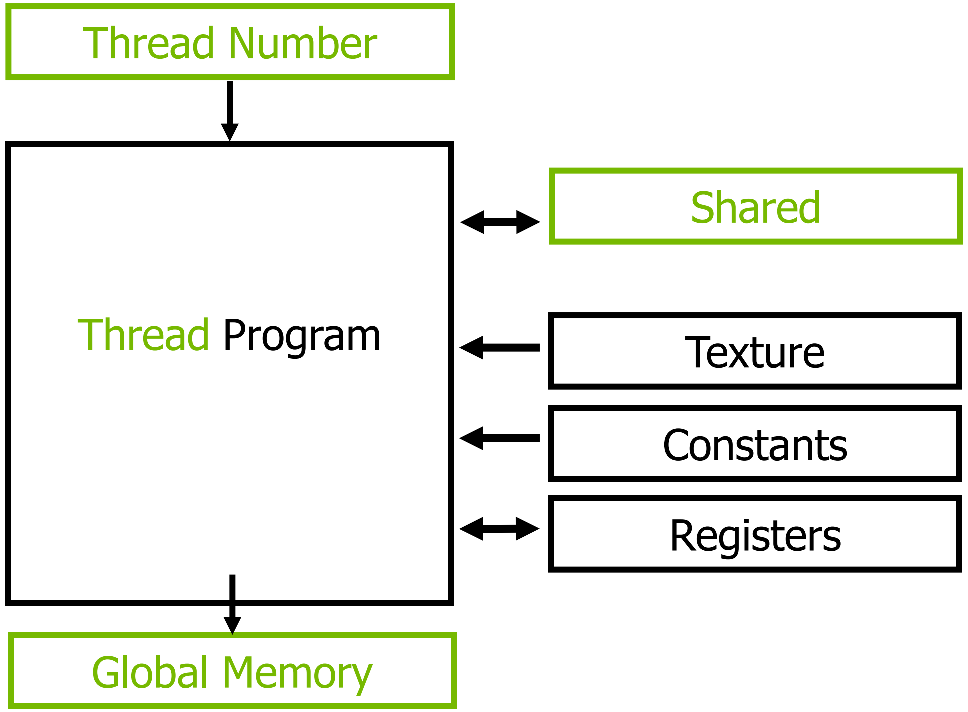


Features

- Fully general load/store to GPU memory: Scatter/Gather
- Programmer flexibility on how memory is accessed
- Untyped, not limited to fixed texture types
- Pointer support

GPU TECHNOLOGY CONFERENCE

Shared Memory

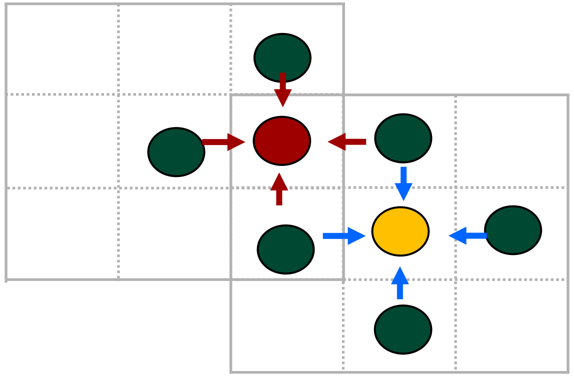


Features

- Dedicated on-chip memory
- Shared between threads for inter-thread communication
- Explicitly managed
- As fast as registers

Example Algorithm - Fluids

Goal: Calculate PRESSURE in a fluid



Pressure depends on neighbors

Pressure = Sum of neighboring pressures

$P_n' = P_1 + P_2 + P_3 + P_4$

So the pressure for each particle is…

$Pressure_1 = P_1 + P_2 + P_3 + P_4$

$Pressure_2 = P_3 + P_4 + P_5 + P_6$

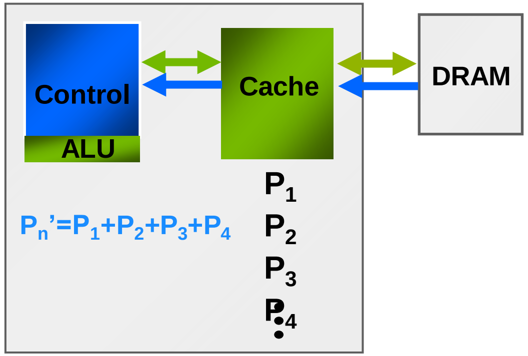
$Pressure_3 = P_5 + P_6 + P_7 + P_8$

$Pressure_4 = P_7 + P_8 + P_9 + P_{10}$

⋮

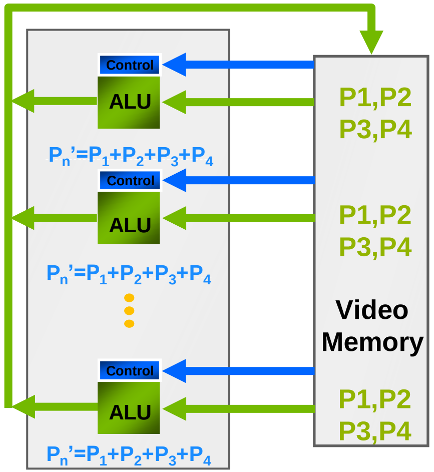
Example Fluid Algorithm

CPU



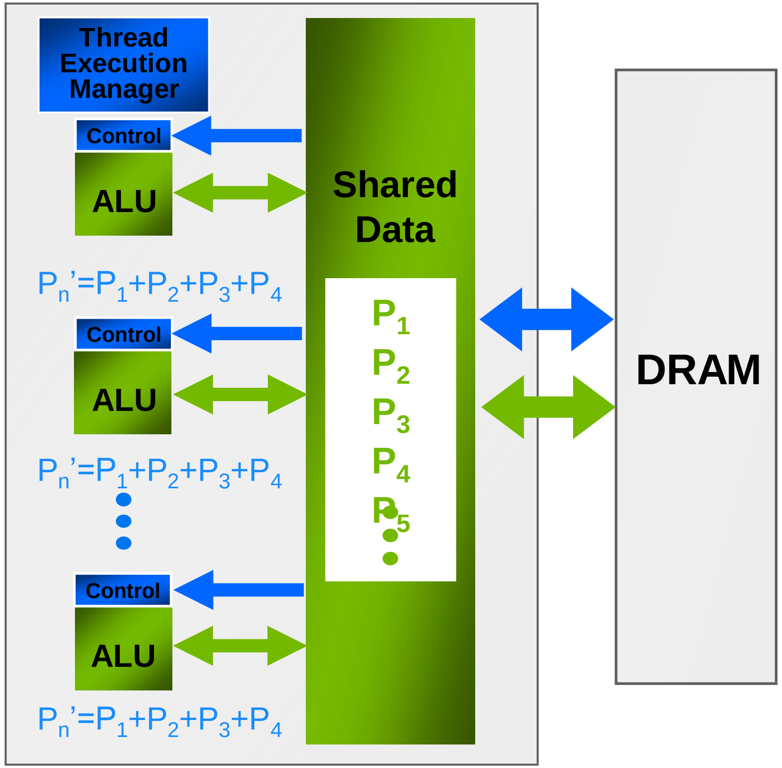
Single thread out of cache

GPGPU



Multiple passes through video memory

CUDA GPU Computing

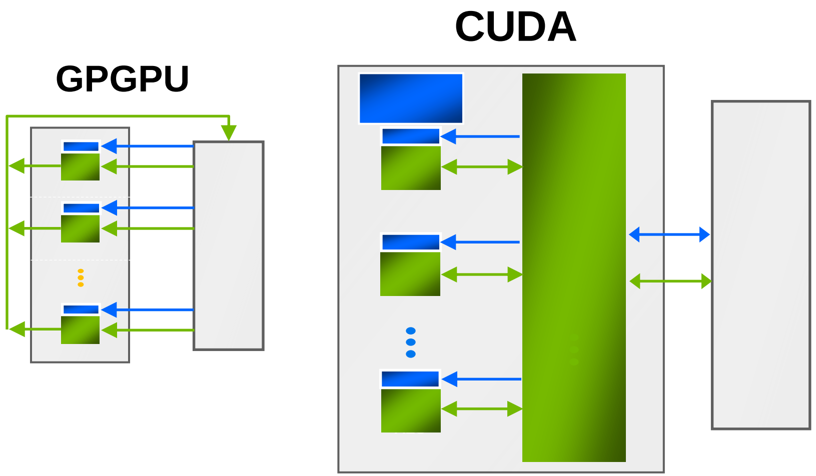


Data/Computation

Program/Control

Streaming vs. GPU Computing

- **Streaming**
  - Gather in, Restricted write
  - Memory is far from ALU
  - No inter-element communication
- **CUDA**
  - More general data parallel model
  - Full Scatter / Gather
  - PDC brings the data closer to the ALU
  - App decides how to decompose the problem across threads
  - Share and communicate between threads to solve problems efficiently



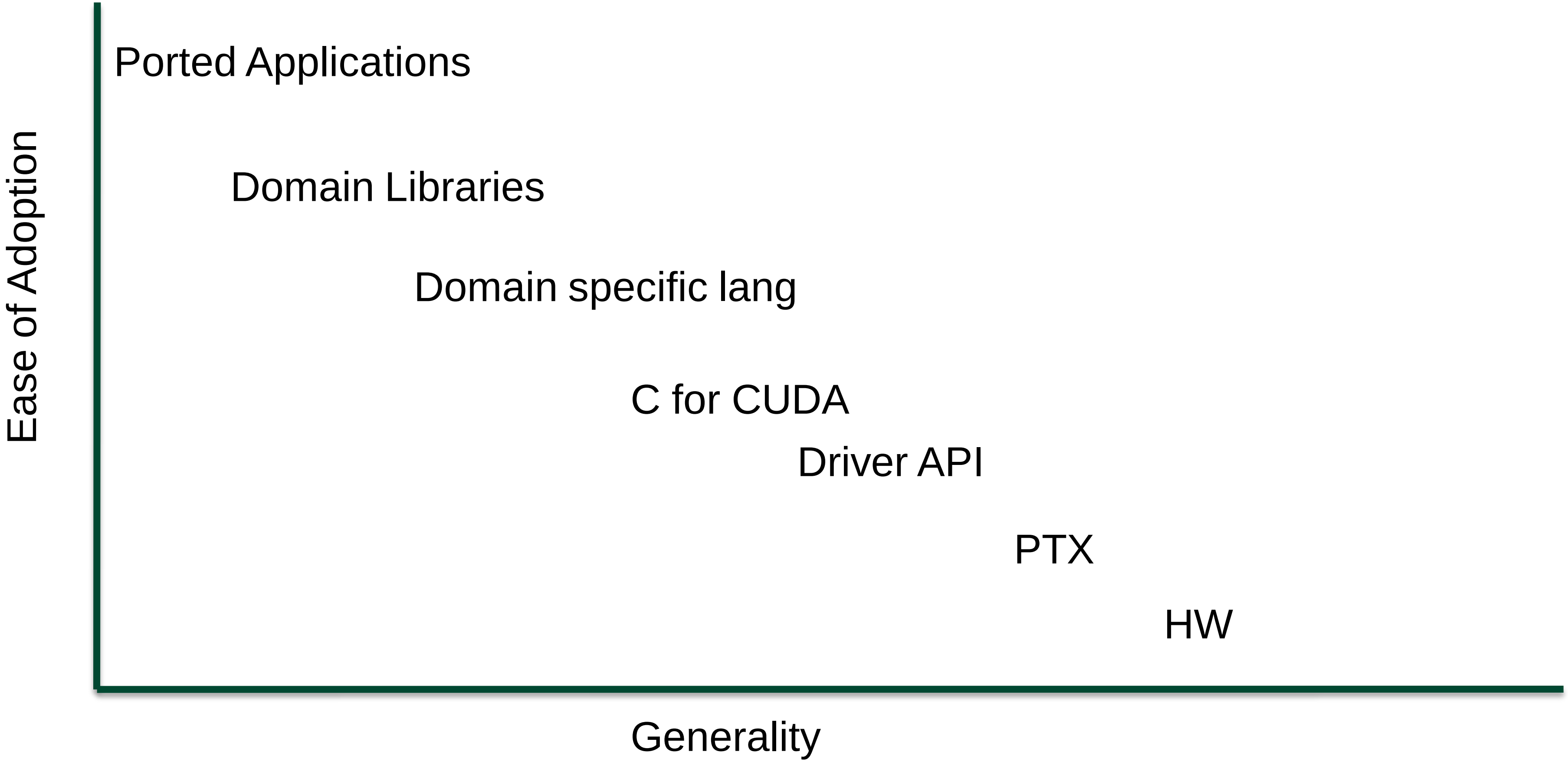
Divergence in Parallel Computing

- Removing divergence pain from parallel programming
- SIMD Pain
  - User required to SIMD-ify
  - User suffers when computation goes divergent
- GPUs: Decouple execution width from programming model
  - Threads can diverge freely
  - Inefficiency only when granularity exceeds native machine width
  - Hardware managed
  - Managing divergence becomes performance optimization
  - Scalable

CUDA: Threading in Data Parallel

- Threading in a data parallel world
  - Operations drive execution, not data
- Users simply given thread id
  - They decide what thread access which data element
  - One thread = single data element or block or variable or nothing….
  - No need for accessors, views, or built-ins

Customizing Solutions



Ahead of the Curve

- GPUs are already at where CPU are going
- Task parallelism is short lived...
- Data parallel is the future
  - Express a problem as data parallel....
  - Maps automatically to a scalable architecture
- CUDA is defining that data parallel future

BACKUP

Stunning Graphics Realism

Lush, Rich Worlds

Incredible Physics Effects

Core of the Definitive Gaming Platform

GPGPU Programming Model

OpenGL Program to Add A and B

↓

Vertex Program

↓

Rasterization

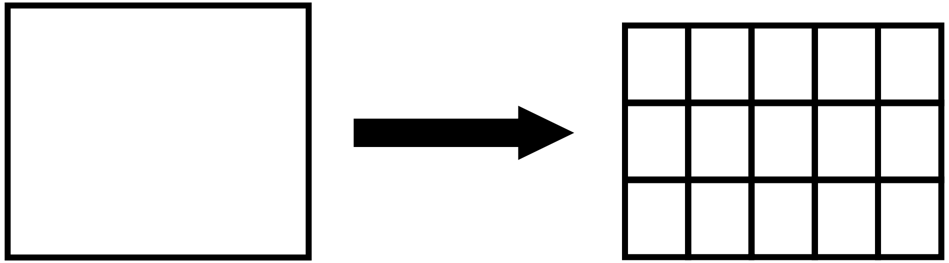
↓

Fragment Program

↓

CPU Reads Texture Memory for Results

"Programs" created with raster operation



Read textures as input **to OpenGL shader program**

Write answer to texture memory as a "color"