# Automatic Tuning Matrix Multiplication Performance on Graphics Hardware

Changhao Jiang (cjiang@cs.uiuc.edu)

Lecture notes for CS498dp Fall 2006
University of Illinois Urbana Champaign

## Outline

- Introduction to GPGPU
- GPU architecture
- Programming model and the Brook language for GPU
- Automatic matrix multiply library generation on GPU
- Performance results

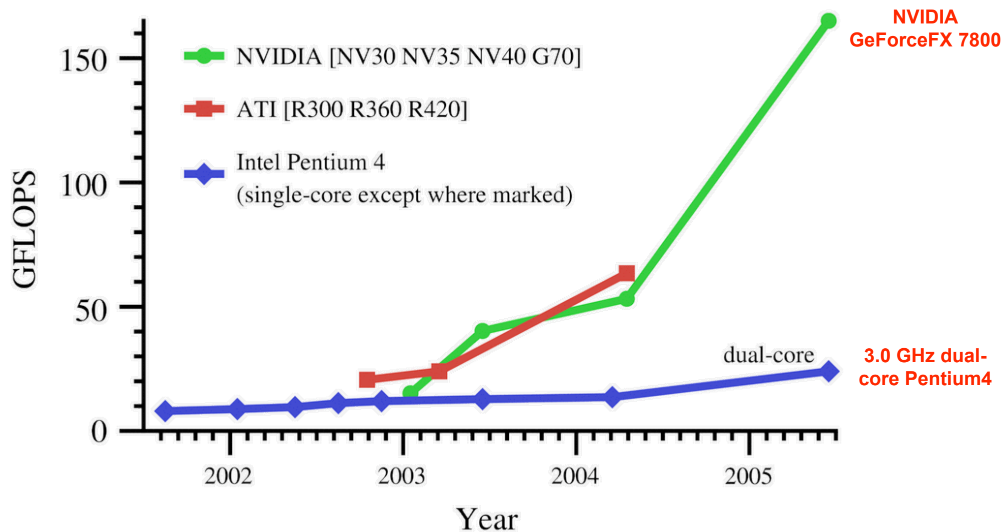
## Introduction

- The GPU on commodity video cards has evolved into an extremely flexible and powerful processor
  - Programmability
  - Precision
  - Power
- This class will introduce how to harness GPU power and how to automatically tune matrix multiplication performance on GPU
  - an ATLAS for GPU ☺

## Motivation: Computational Power

- GPUs are fast…
  - 3.0 GHz dual-core Pentium4: 24.6 GFLOPS
  - NVIDIA GeForceFX 7800: 165 GFLOPs
  - 1066 MHz FSB Pentium Extreme Edition : 8.5 GB/s
  - ATI Radeon X850 XT Platinum Edition: 37.8 GB/s
- GPUs are getting faster, faster
  - CPUs: 1.4× annual growth
  - GPUs: 1.7×(pixels) to 2.3× (vertices) annual growth

# GPU becomes more powerful




NVIDIA GeForceFX 7800

3.0 GHz dual-core Pentium4

*Courtesy Ian Buck, John Owens*

# An Aside: Computational Power

- *Why are GPUs getting faster so fast?*
  - Arithmetic intensity: the specialized nature of GPUs makes it easier to use additional transistors for computation instead of cache
  - Economics: multi-billion dollar video game market is a pressure cooker that drives innovation

## Motivation: Flexible and Precise

- Modern GPUs are deeply programmable
  - Programmable pixel, vertex, video engines
  - Solidifying high-level language support
- Modern GPUs support high precision
  - 32 bit floating point throughout the pipeline
  - High enough for many (not all) applications

## Motivation: The Potential of GPGPU

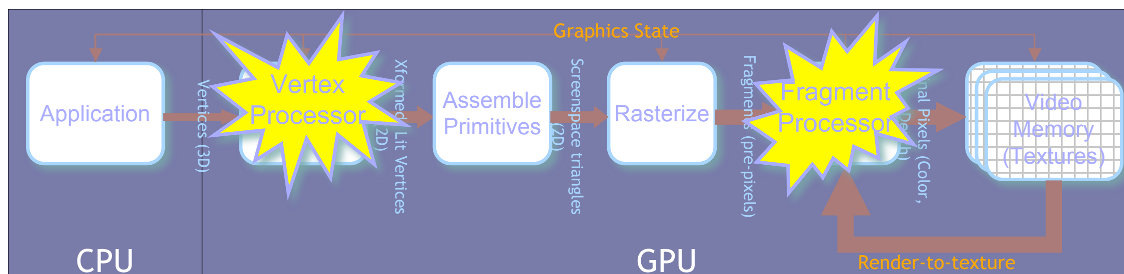
- GPGPU (General purpose computation on GPUs)
  - Active research topic:
    - http://www.gpgpu.org
- In short:
  - The power and flexibility of GPUs makes them an attractive platform for general-purpose computation
  - Example applications range from in-game physics simulation to conventional computational science
  - Goal: make the inexpensive power of the GPU available to developers as a sort of computational coprocessor

## The Problem: Difficult To Use

- GPUs designed for & driven by video games
  - Programming model unusual
  - Programming idioms tied to computer graphics
  - Programming environment tightly constrained
- Underlying architectures are:
  - Inherently parallel
  - Rapidly evolving (even in basic feature set!)
  - Largely secret
- Can't simply "port" CPU code!

## GPU architecture

- Simplified graphics pipeline

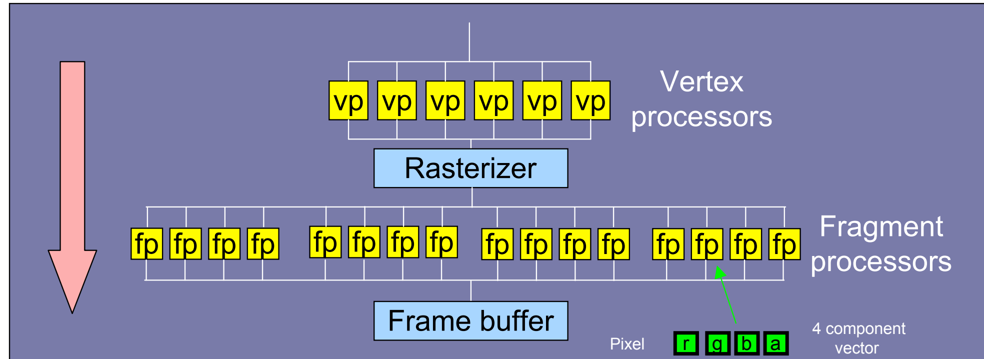



*Courtesy David Luebke*

- Programmability was introduced into two stages

# GPU architecture

- Another view of GPU architecture




- Most horsepower of GPGPU comes from the “fragment processors”
- The same “shader” runs synchronously on all fragment processors
- Every fragment processor can execute SIMD instructions.

# Unusual features/constraints of GPU program

- SIMD instructions with smearing and swizzling
  - *R2.rgba = R1.abgr * R3.ggab*
- Limit on instruction count
  - A shader can contain limited number of instructions
- Limit on output
  - No scatter operations
  - Designated memory locations (limited number)
- Limit on branch instruction on some GPUs
  - “If-then-else” → predicated instructions
  - “loop” → fully unrolled

## High Level Shading Languages

- Cg, HLSL, & OpenGL Shading Language
  - Cg:
    - http://www.nvidia.com/cg
  - HLSL:
    - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/highlevellanguageshaders.asp
  - OpenGL Shading Language:
    - http://www.3dlabs.com/support/developer/ogl2/whitepapers/index.html
- Requires knowledge of computer graphics and graphics API (OpenGL or DirectX)

## GPGPU Languages

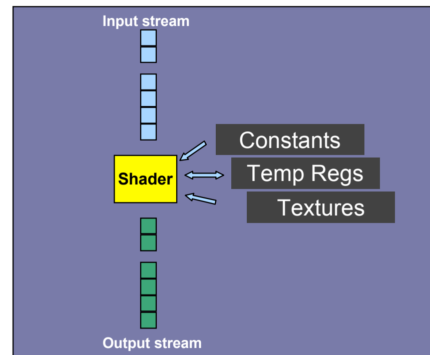
- Why do we want them?
  - Make programming GPUs easier!
    - Don't need to know OpenGL, DirectX, or ATI/NV extensions
    - Simplify common operations
    - Focus on the algorithm, not on the implementation
- Sh

  University of Waterloo
  http://serioushack.com
  http://libsh.sourceforge.net
- Brook

  Stanford University
  http://brook.sourceforge.net
  http://graphics.stanford.edu/projects/brookgpu

## Brook: General Purpose Streaming Language

- Stream programming model
  - The same kernel program (shader) operates on streams of data.
- C with stream extensions
- Cross platform
  - ATI & NVIDIA
  - OpenGL & DirectX
  - Windows & Linux



## 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

```
streamRead (r, r_ptr);
streamWrite (velocityfield, v_ptr);
```

# Kernels

- Functions applied to streams
  - similar to for_all construct
  - no dependencies between stream elements

```
kernel void foo (float a<>, float b<>,
                 out float result<>) {
   result = a + b;
}

float a<100>;
float b<100>;
float c<100>;

foo(a,b,c);
```

```
for (i=0; i<100; i++)
       c[i] = a[i]+b[i];
```

# Reductions

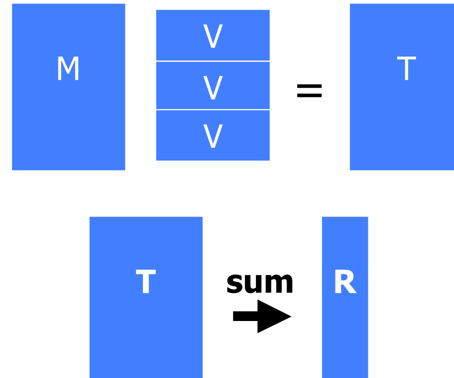
- Compute single value from a stream
  - associative operations only

```
reduce void sum (float a<>,
                 reduce float r<>)
   r += a;
}

float a<100>;
float r;

sum(a,r);
```

```
r = a[0];
for (int i=1; i<100; i++)
  r += a[i];
```

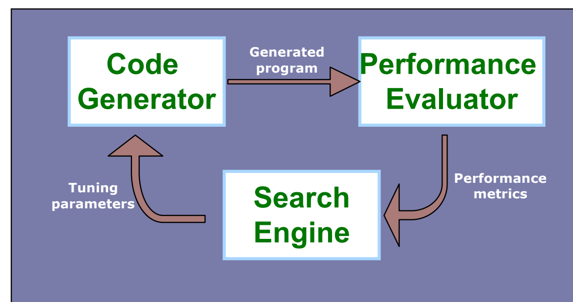
# Matrix Vector Multiply

```
kernel void mul (float a<>, float b<>,
                         out float result<>) {
   result = a*b;
}

reduce void sum (float a<>,
                         reduce float result<>) {
   result += a;
}

float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);
```



# An automatic matrix-multiply generation system

- An automatic matrix multiply generation system, which includes:
  - A code generator:
    - Generate multiple versions in high level BrookGPU language, which will be compiled into low level code.
  - A search engine:
    - Searches in the implementation space for the best version
  - A performance evaluator:
    - Measure performance of generated code

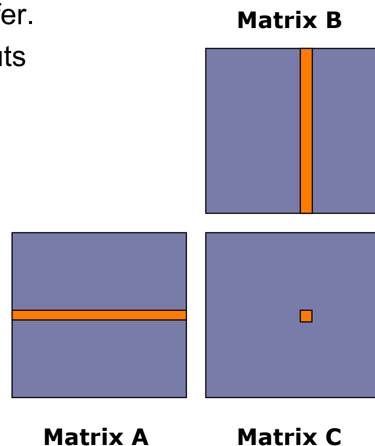


# Code generator

```
$ python codegen.py -h

   codegen [options]
     options :
      -h, --help
           print this help
      -u, --unroll
           unroll the kernel function
           (defulat not unroll)
      -p, --np=[passes]
           the number of iterations executed in each invocation of kernel
           default 1
      -m, --mrt_height=[height]
           the height of multiple-render-target(mrt) grid
           default 1
      -n, --mrt_width=[width]
           the width of multiple-reder-target(mrt) grid
           default 1
      -c, --channel=[channel code]
           how to use multiple (up to 4) color channels of a fragment
              1 : 1 x 1 (default)
              2 : 1 x 2
              3 : 1 x 3
              4 : 1 x 4
              5 : 2 x 2
           default value "1"
```

# GPU algorithms for matrix multiply

- Straightforward mapping of triply nested loop onto GPU
  - Store two input matrices (A and B) as two textures
  - Store the resulting matrix C in the frame buffer.
  - Each execution of the shader program outputs one element of C
    - Fetches one row from matrix A
    - Fetches one column from matrix B
    - Computes the dot product. Save result to C
- Problems:
  - No data reuse in the shader
    => poor performance
  - Shader length might exceed instruction limit if loop is unrolled due to the lack of branch instruction



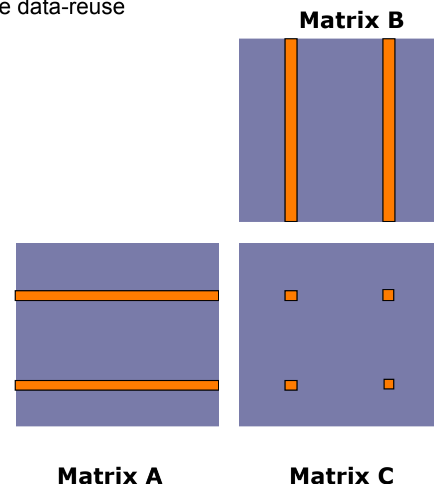
# Generated code:

- python codegen.py –p 1024

```
for (start =0.0f ;start<1024/1;start +=1024) {
    mult(index, a, b, start, sum0, c0);
    streamSwap(sum0, c0);
  }

kernel void mult(iter float2 index<>, float  a[][], float  b[][], float start, float  sum0[][], out float  c0<>){
   float i;
   float2 addr_a;
   float2 addr_b;
   float2 tmp_addr;
   float  a_tmp00;
   float  b_tmp00;
   c0 = sum0[index];

   addr_a = float2(start,index.y);
   addr_b = float2(index.x, start*1);

   for (i=0.0f; i<1024; i=i+1.0f) {
 // load a column/row from texture(matrix) a
     tmp_addr = addr_a;
     a_tmp00 = a[tmp_addr];

     tmp_addr = addr_b;

      // load an element from texture(matrix) b
     b_tmp00 = b[tmp_addr];
     c0 += a_tmp00  * b_tmp00;

     addr_a.x += 1.0f ; addr_b.y += 1.0f*1;
   }
}
```

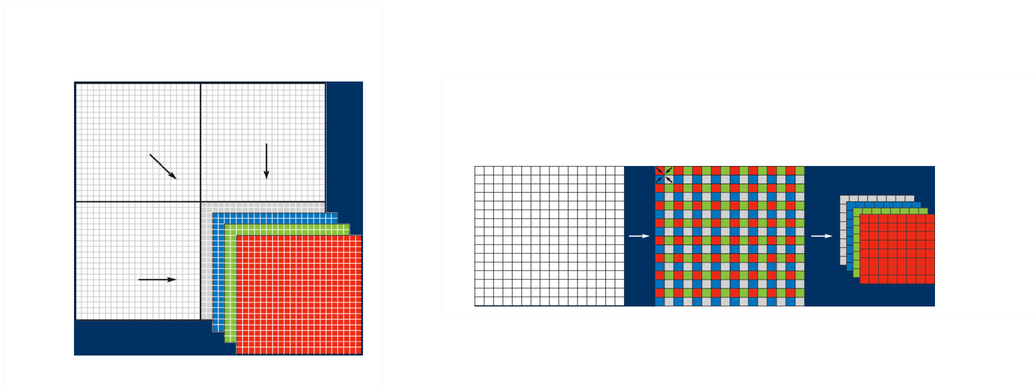
# Tuning for multi-render-targets

- “multi-render-targets”:
  - allows a shader to simultaneously write to multiple buffers
- Tuning strategy:
  - Take advantage of “multi-render-targets” to improve data-reuse
- Algorithm with multi-render-targets:
  - Divide matrix C into mxn sub matrix blocks
    - Each of them will be a render-target
    - A and B are logically divided too
  - Each fragment program
    - Fetches m rows from matrix A
    - Fetches n columns from matrix B
    - Computes mxn dot products
- Downside:
  - The shader require more temporary registers
  - Using multi-render-target has performance overhead
- Code generation command:
  - “python codegen.py –m 2 –n 2”



# Data packing

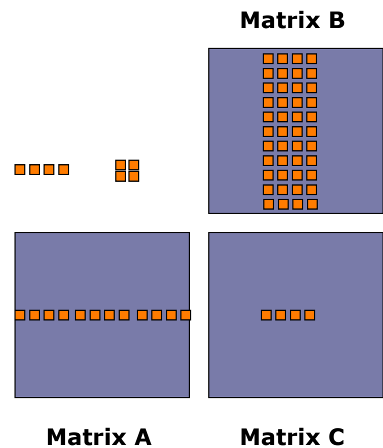
- Pack scalar data into RGBA in texture memory



# Tuning for SIMD instruction with data packing

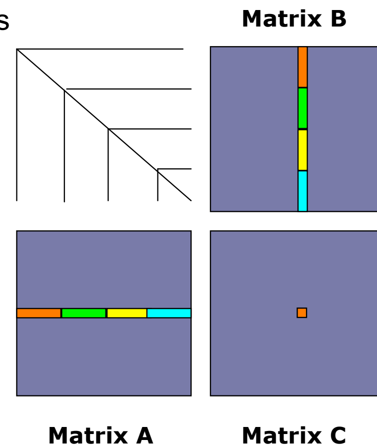
- Fragment processor supports SIMD instructions
- Tuning strategy:
  - Use SIMD instruction to improve performance
  - Use smearing and swizzling to do "register tiling" to improve data reuse
- Algorithm of tuning for SIMD instruction with data packing
  - Packing four elements into one pixel
    - Two schemes: 1x4 vs. 2x2
  - Each fragment program (1x4 scheme)
    - Fetches one row from matrix A
    - Fetches four columns from matrix B
    - Perform a series of vector by matrix product
- Question:
  - What packing scheme is the best in performance?
- Code generation command:
  - "python codegen.py –c 1"

  -c, --channel=[channel code]
  1 : 1 x 1 (default)
  2 : 1 x 2
  3 : 1 x 3
  4 : 1 x 4
  5 : 2 x 2



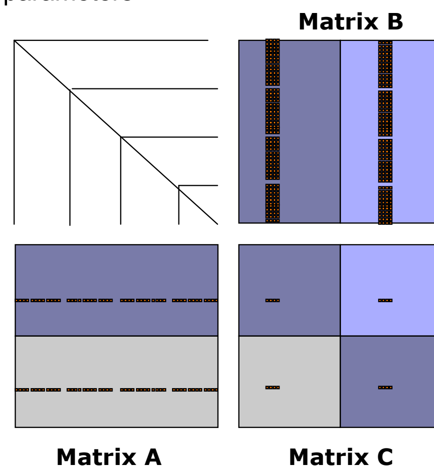
# Tuning the number of passes

- Problem:
  - GPU's limit on instruction count prevents the dot product to be completed in one pass
- Strategy:
  - Partition the computation into multiple passes
- Algorithm with multiple passes:
  - Each fragment program
    - Fetches a part of a row from matrix A
    - Fetches a part of a column from matrix B
    - Perform a dot product to get a partial sum
  - Iterate multiple times to get the final result
- Downside
  - Multi-pass results in expensive overhead in copying intermediate results
- Code generation command:
  - "python codegen.py –p 32"



# Tuning parameters

- The code generator is controlled by a set of tuning parameters
- Tuning parameters
  - "mrt_w", "mrt_h"
    - How to divide matrix C
  - "mc_w", "mc_h"
    - How to pack data to use SIMD
  - "np"
    - How many iterations executed in each pass
  - "unroll"
    - Whether or not to use branch instructions
  - "compiler"
    - To use "cgc" or "fxc" compiler
  - "shader"
    - To use DirectX backend with "ps20", "ps2a", "ps2b", "ps30", or use OpenGL backend with "arbfp", "fp30", "fp40"



# Search strategy

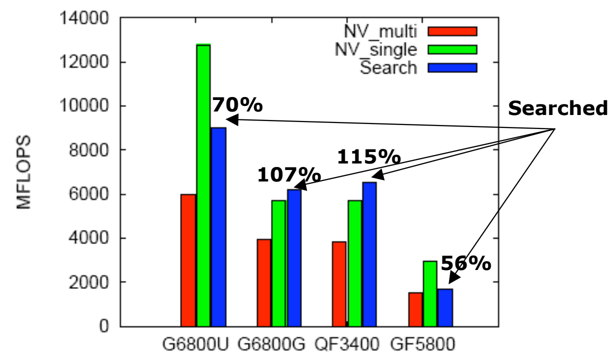
- Search in an exponential space is time-consuming.
- Two techniques employed to speed up the search
  - Space pruning
    - Limit the search range of parameters based on problem-specific heuristics
  - Search in phases
    - Search parameters in phases
    - Search order:

1: For each *compiler* value
2: For each *profile* value
3: For each *unroll* value
4: Search *np* in power of two values
5: For each *mc_** value
6: For each *mrt_** value
7: Evaluate Performance
8: Recursively search *np* in both sides of best *np* found in step 4.

- The search time reduces dramatically
  - from 53 days in theory to 4 hours in practice, with no significant performance loss.

# Performance data

- Compare with two expert hand-tuned implementations
  - Part of GPUBench developed at Stanford University
  - Implemented with carefully crafted assembly code
- Comparable performance on four GPU platforms
  - On two platforms
    - beats hand-tuned by **8%** and **15%**
  - On the other two platforms
    - achieves **56%** and **70%** of hand-tuned version.



# Performance penalties of using a high level language

- One reason for lower performance than manual tuning:
  - Overhead in using the high-level BrookGPU language.
- Compare the performance of the same algorithm implemented in
  - BrookGPU
  - Assembly code