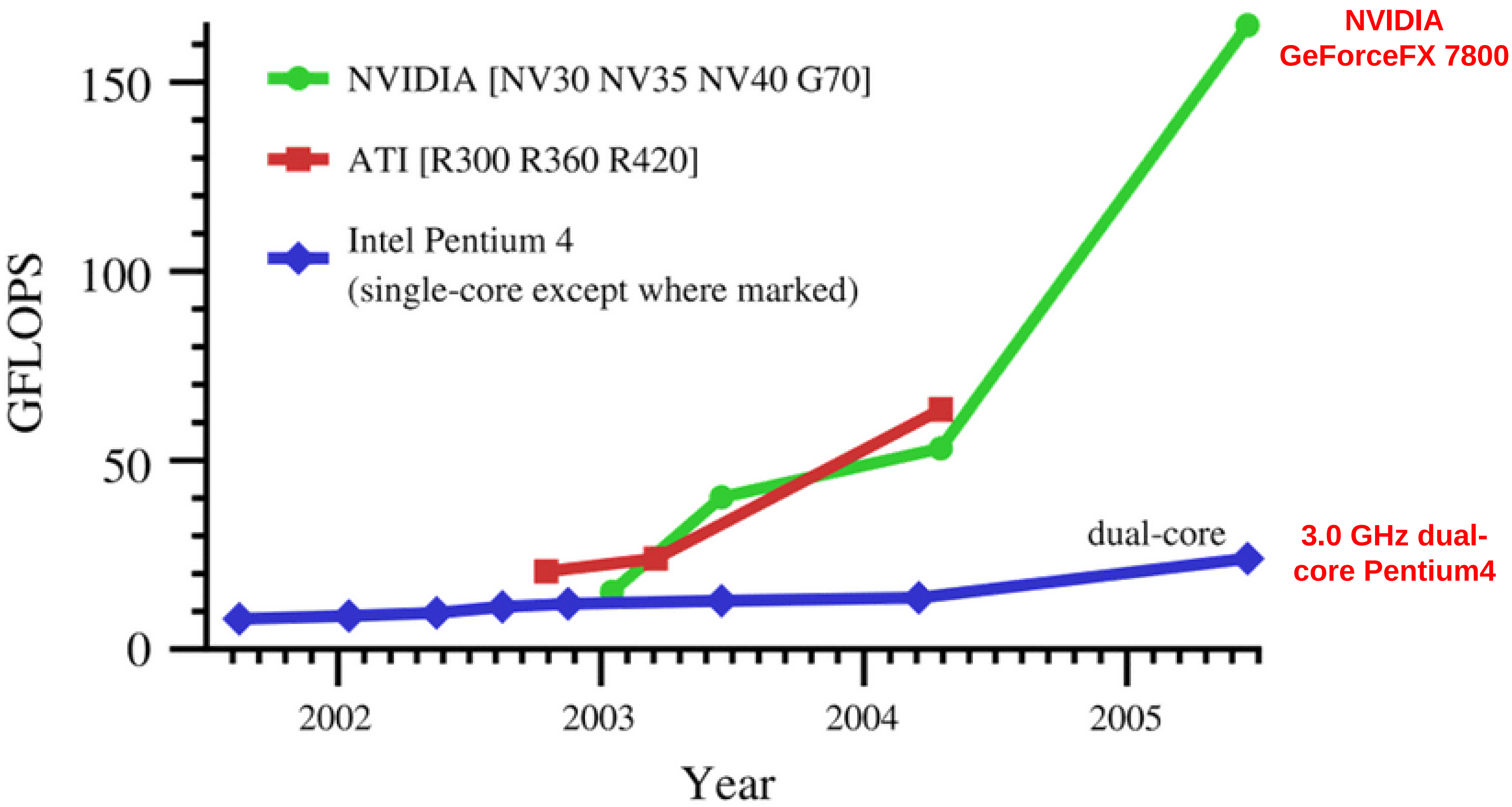
Automatic Tuning Matrix Multiplication Performance on Graphics Hardware

Changhao Jiang (cjiang@cs.uiuc.edu)
Marc Snir (snir@cs.uiuc.edu)

University of Illinois Urbana Champaign

GPU becomes more powerful



Courtesy Ian Buck, John Owens

Use of GPU for non-graphics applications

- GPGPU (General purpose computation on GPUs)
  - Goal: make the flexible and powerful GPU available to developers as a computational coprocessor.
- Difficulties of GPGPU
  - Unusual programming model
  - Programming idioms tied to computer graphics
  - Underlying architecture evolves fast
  - Architecture internals largely secret

Automatic library generation system

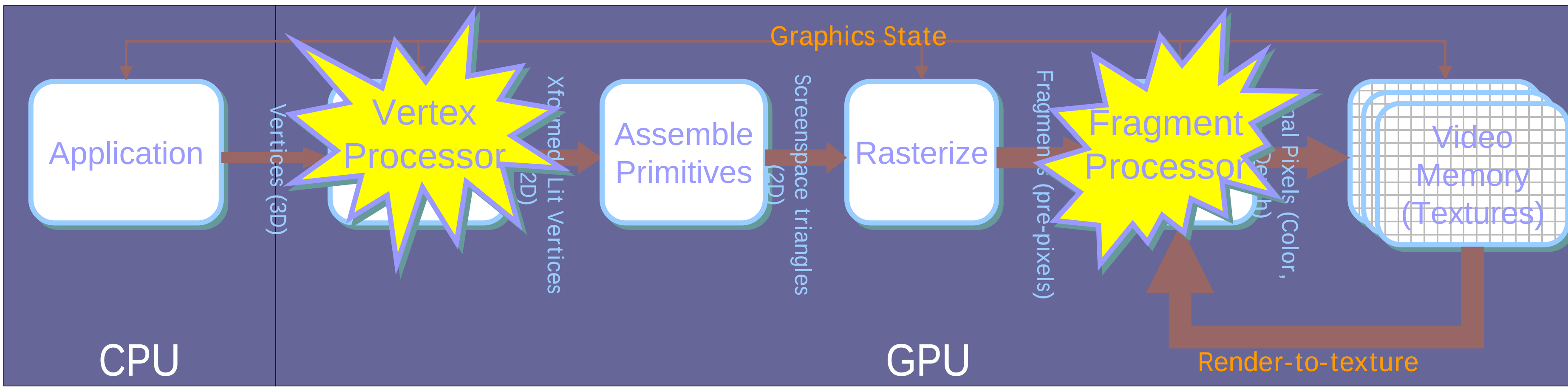
- Automatic library generation can help
  - Generate high-performance libraries by empirical search
- Successful example systems on CPUs:
  - ATLAS
    - *Whaley, Petitet, Dongarra*
  - Sparsity
    - *Im, Yelick, Vuduc*
  - FFTW
    - *Frigo, Johnson*
  - Spiral
    - *Puschel, Singer, Xiong, Moura, Johnson, Padua*
  - Adaptively tuned sorting library
    - *Li, Garzaran, Padua*

Our work

- Implemented a high-performance matrix multiplication library generator for GPU
  - An “ATLAS for GPU”
- Main contributions:
  - The first automatic library generation system for GPUs
  - Identifies several tuning strategies unique for GPUs
  - Implements a customized search-engine
  - The automatically generated code has comparable performance with expert manually tuned version.

GPU architecture

- Graphics pipeline

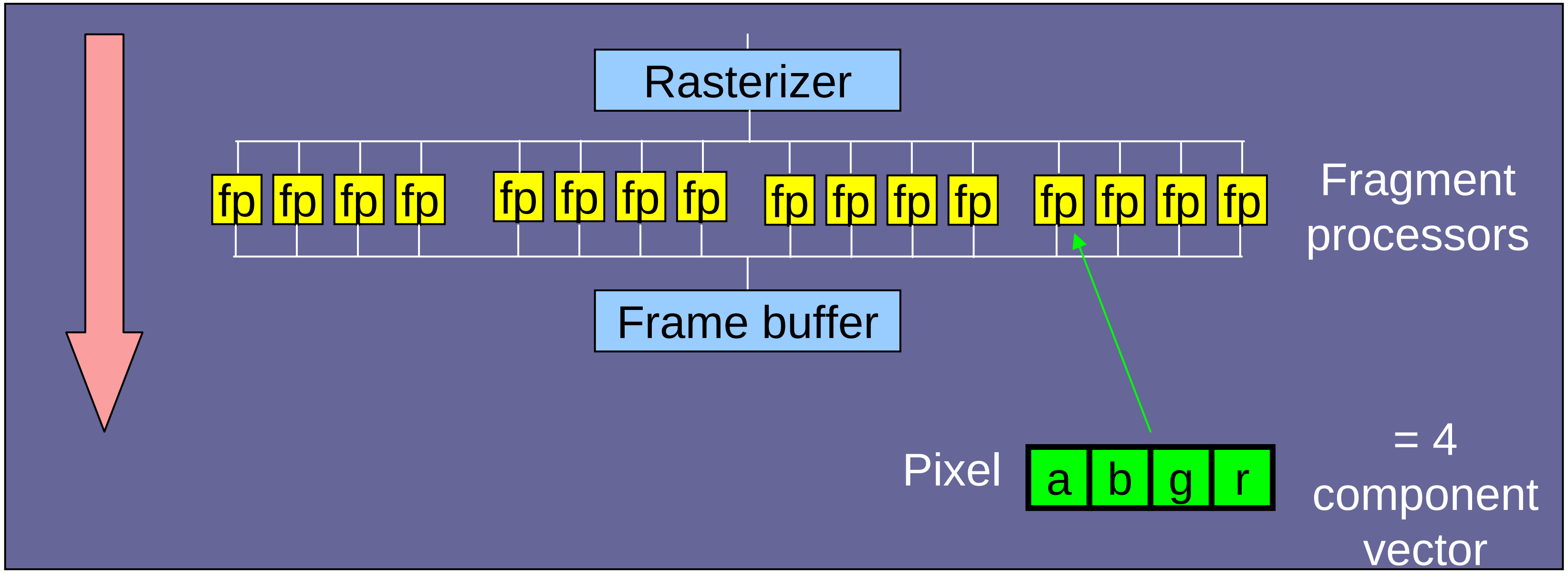


Courtesy David Luebke

- Programmability was introduced into two stages

GPU architecture

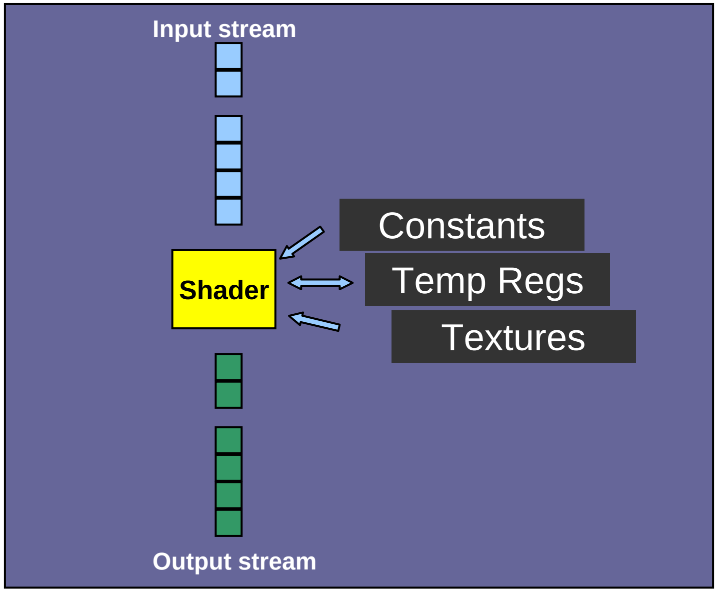
- Another view of GPU architecture



- 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 on the 4 channels of a pixel.

GPU programming model

- Stream processing model
  - The same kernel program (shader) operates on streams of data.

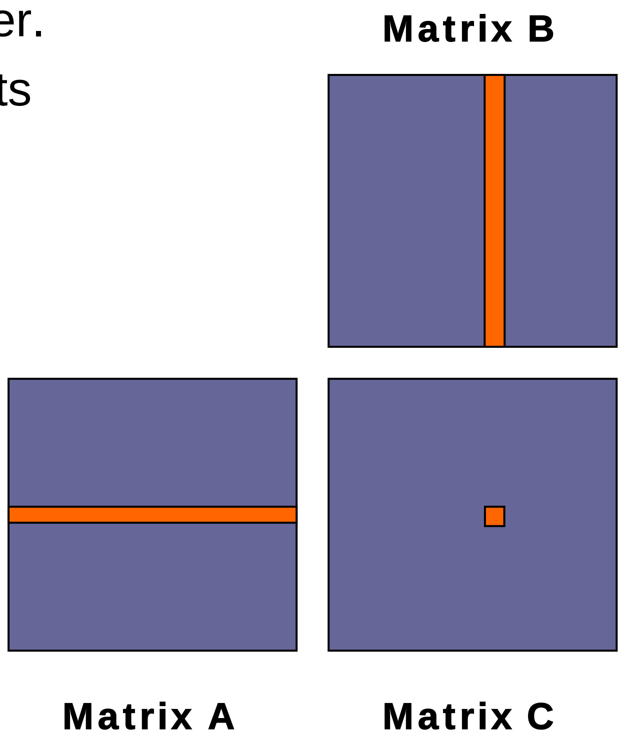


Unusual features/constraints of GPU program

- SIMD instructions with smearing and swizzling
  - *R2 = R1.abgr * R3.ggab*
- Limit on instruction count
- Limit on output
- Limit on branch instruction

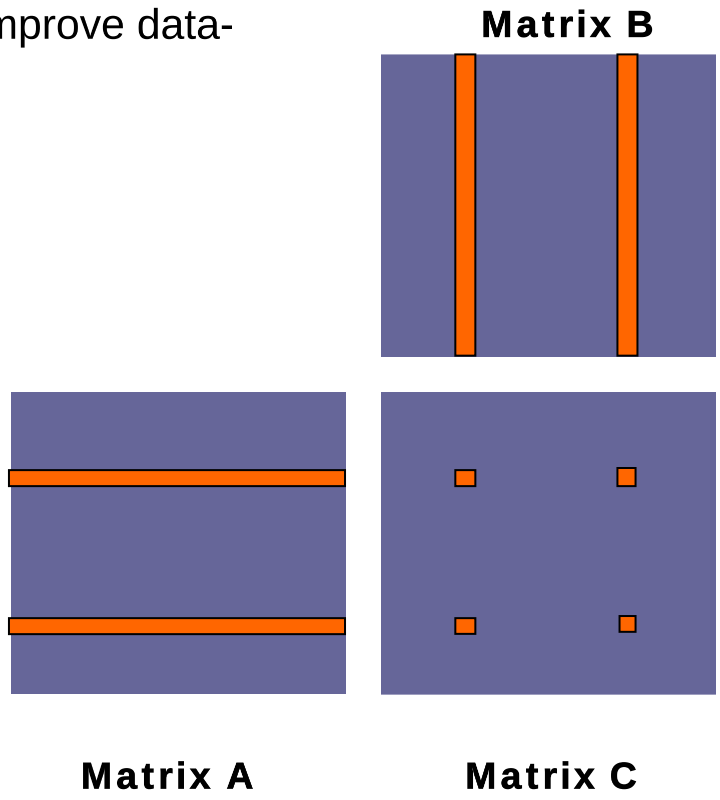
GPU algorithms for matrix multiply

- 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



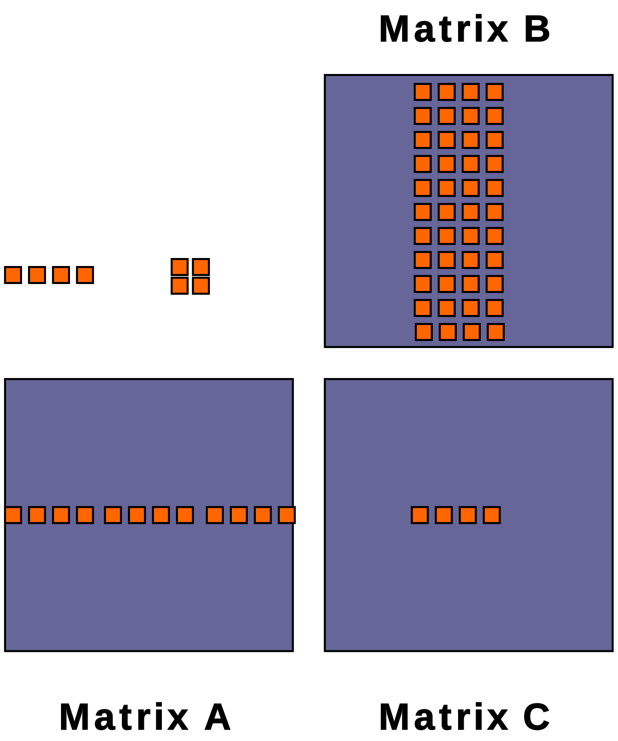
Tuning for multi-render-targets

- “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



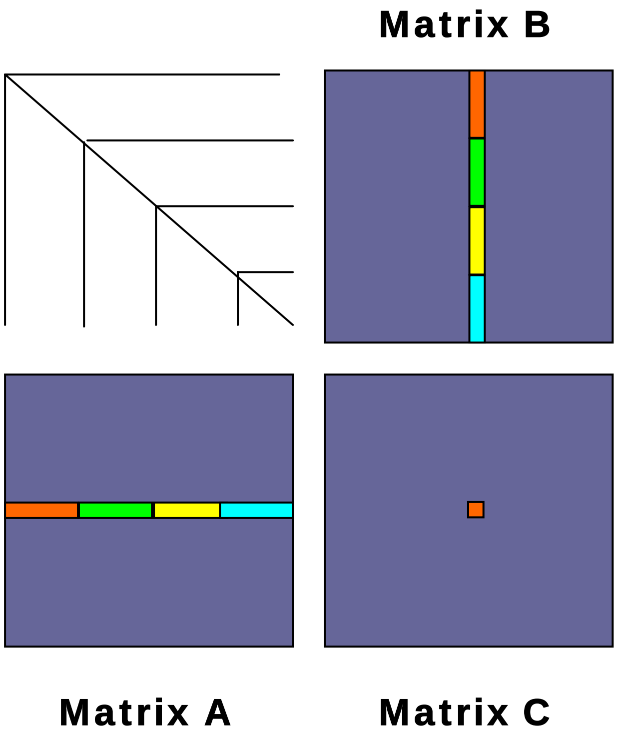
Tuning for SIMD instruction with data packing

- 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?



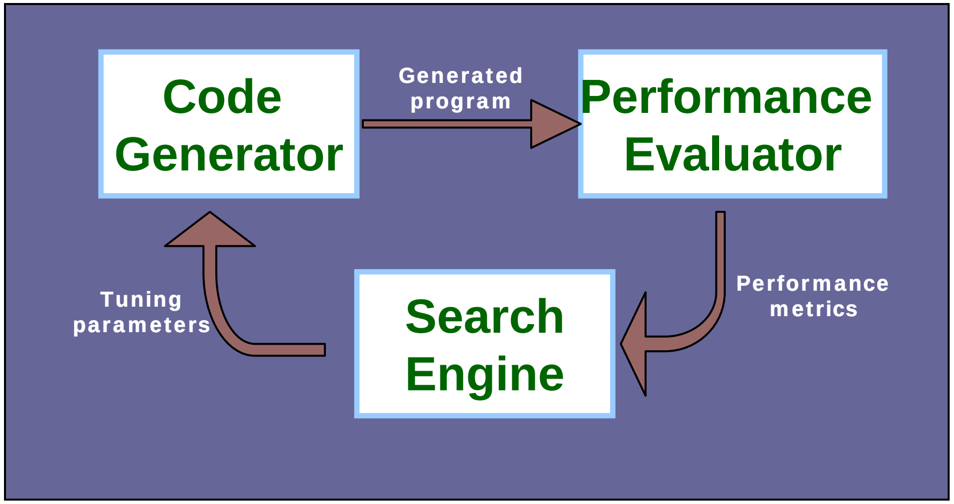
Tuning the number of passes

- 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



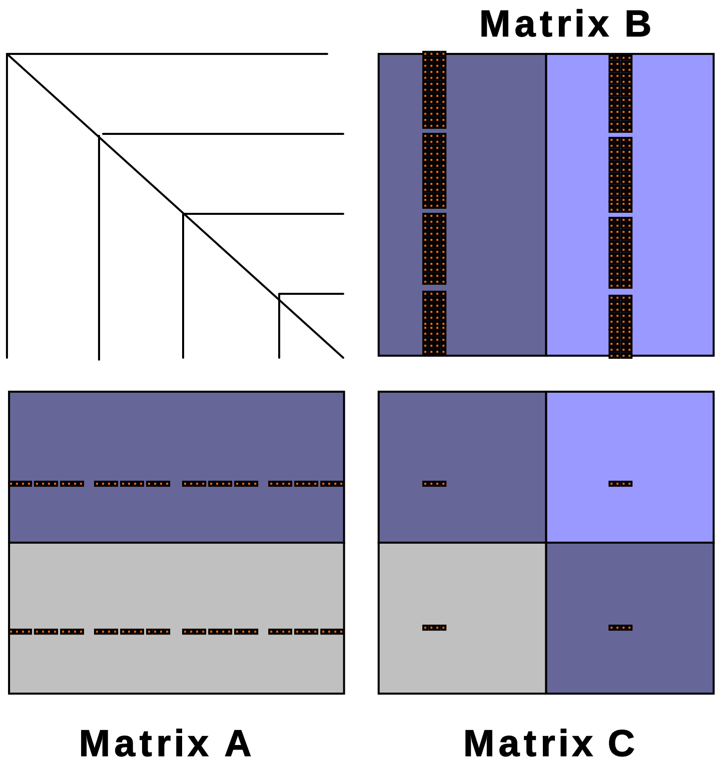
An automatic matrix-multiply generation system

- 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 implementa-tion space for the best version
  - A performance evaluator:
    - Measure performance of generated code



Tuning parameters

- Generated code combines the previous tuning strategies
- 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:

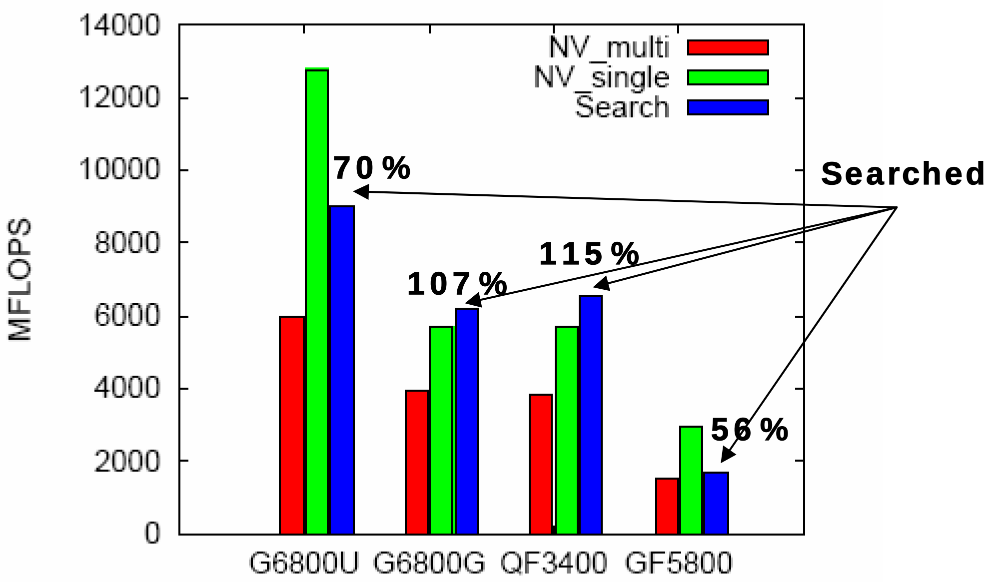
1: For each *compiler* value
2: For each *profile* value
3: For each *unroll* value
4: Search *np* in power of two values
5: For each *mc_** value
6: For each *mrt_** value
7: Evaluate Performance
8: Recursively search *np* in both sides of best *np* found in step 4.

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

Performance data

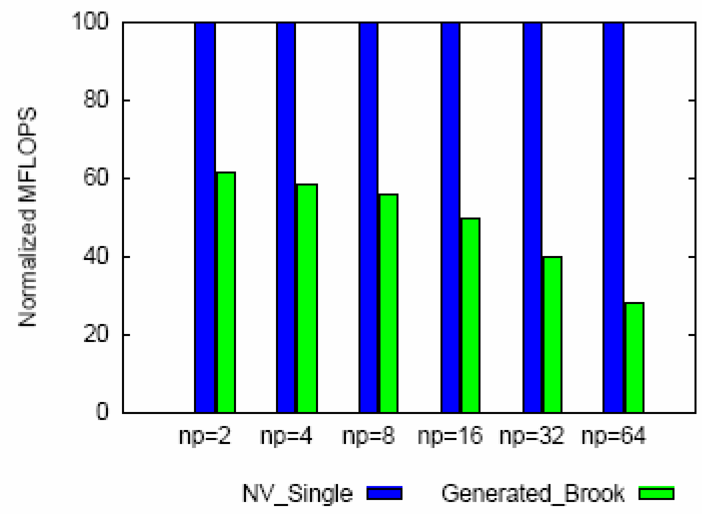
- Compare with two expert hand-tuned implementations
  - Part of GPUBench developed at Stanford University
  - Implemented with carefully crafted assembly code

- 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



Potential future research directions

- Improve high-level BrookGPU's performance
- Generating more libraries for GPU
  - Signal processing (FFT)
  - Numerical libraries
  - Sorting library