ME 290-R:

General Purpose Computation (CAD/CAM/CAE) on the GPU

(a.k.a. Topics in Manufacturing)

Sara McMains

Spring 2009

Lecture 18: Matrix Multiply Tuning

Automatic Tuning Matrix Multiplication Performance on Graphics Hardware

C. Jiang and M. Snir

PACT, 2005

Table of Contents

- Introduction to matrix multiply
- GPU techniques
- Performance results
- Automated performance tuning

Matrix multiplication is important

- Matrix-matrix multiply is:
  - Basic building block in scientific computing
  - Included in BLAS Level 3
    - BLAS is a cornerstone of dense algebra libraries

- Need dense solvers on GPU?
  - Implement GPU BLAS first!

* (BLAS is Basic Linear Algebra Subprograms)

What is Matrix-Matrix Multiply?

Given matrices $A^{n \times n}$, $B^{n \times n}$, where

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{pmatrix} \qquad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2n} \\ b_{31} & b_{32} & b_{33} & \dots & b_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & b_{n3} & \dots & b_{nn} \end{pmatrix}$$

Find the product $C = AB$ as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Making High Performance Multiply

- Mathematics not too complicated
  - Simpler than Gaussian elimination!
- Can concentrate on performance!

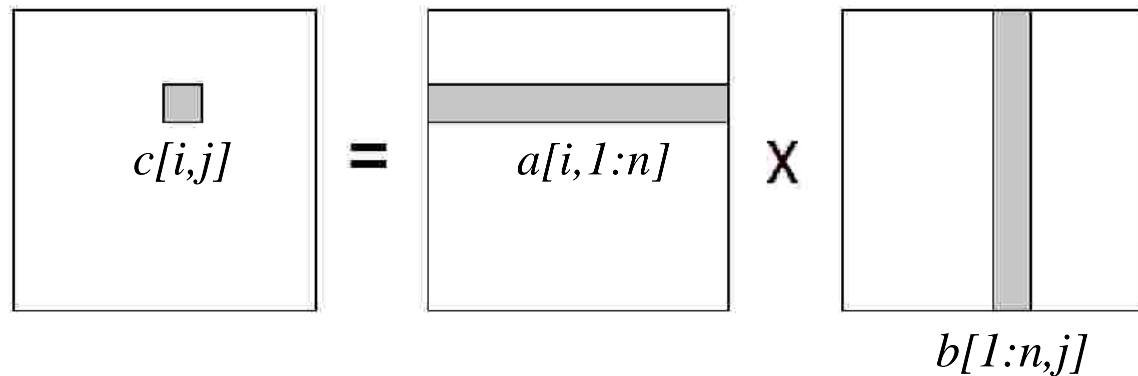
Basic facts about matrix multiply

- Bandwidth-bound
  - $2n^3$ FLOPS
  - $2n^3$ memory references
    - 1 FLOP per memory reference!
    - In fact, all dense linear algebra is bandwidth-bound
- $O(n)$ data reuse
  - input data is only $2n^2$
    - High cache efficiency is the key

Naïve implementation

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            C[i,j] += A[i,k]*B[k,j]
```



What about same on GPU?

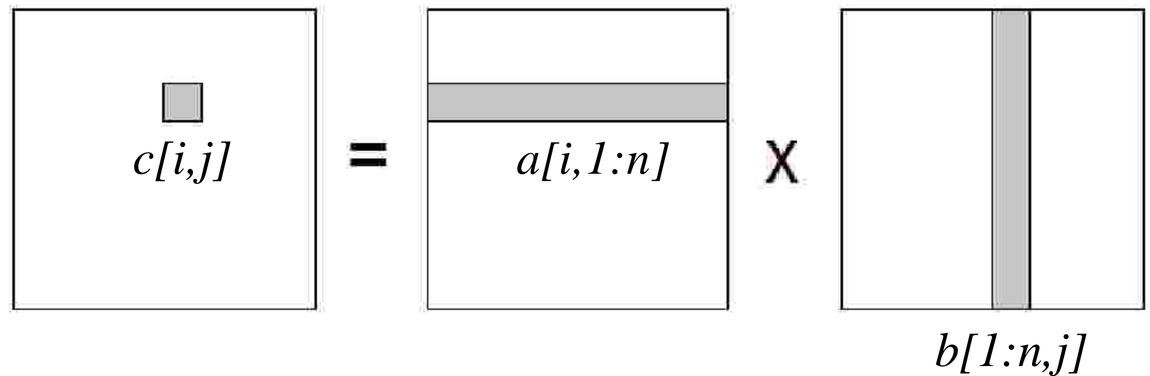
What if matrices were textures

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            C[i,j] += A[i,k]*B[k,j]
```

Render a quad (for i, for j)

Computing `C[i,j]` ⇒ fragment program (for k)

(Fatahalian et al. 2004)

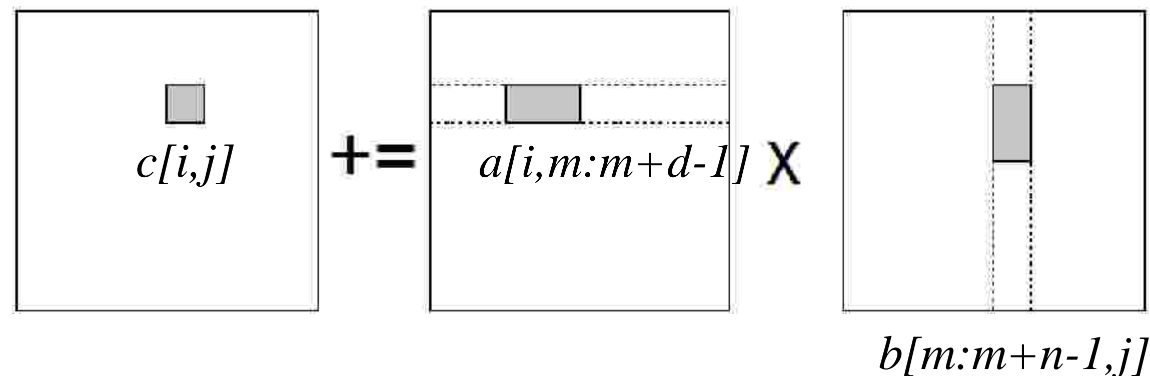


Problem: fragment shader may be too long

Multiple rendering passes

```
for l = 1 to N step d               } N/d rendering passes
    for i = 1 to N                  } Render a quad
        for j = 1 to N              }
            for k = m to m+d-1                    } shorter fragment
                C[i,j] += A[i,k]*B[k,j]           } program
```

(Larsen and McAllister 2001, Hall et al. 2003)



Larger `d` – fewer passes + lower bandwidth requirements

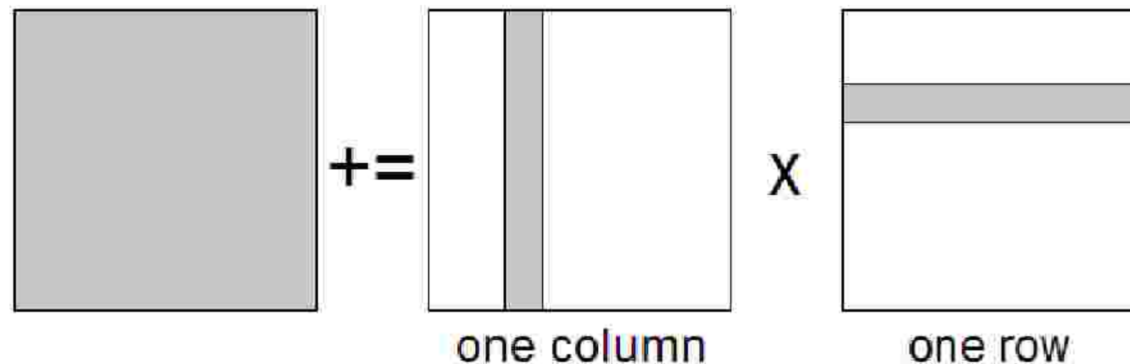
What data is used per pass?

- One pass (for d=1):

  C += outer product of k-th column of A and

  k-th row of B

- Not much data is used per pass
  - Smaller d – more cache efficient



Data use per pass

- Competition on d:
  - Large d are good – lower bandwidth (fewer passes)
  - Large d are bad – may not fit into cache
- Some d in-between should be optimal

Using color channels

- GPU is designed to work with vector data
  - RGBA
  - XYZW
- Example:
  - `R0.rgba += R1.rgba`
  - `R0.r += R1.r`
  - Both are 1 cycle!
- Quadruple FLOPS/s for free!
  - Need cool tricks

2x2 blocking

[Hall, Carr and Hart 2003] :

- Take $N \times N$ matrix
- Think of it as $\mathrm{N}/2 \times \mathrm{N}/2$ matrix of 2x2 blocks
- Each block is RGBA value
- Implement multiplication of blocks

2x2 blocking

2x2 blocks map to RGBA value

$$A.rgba = \begin{pmatrix} A.r & A.g \\ A.b & A.a \end{pmatrix} \qquad B.rgba = \begin{pmatrix} B.r & B.g \\ B.b & B.a \end{pmatrix}$$

Their product is

$$C.rgba = \begin{pmatrix} A.r * B.r + A.g * B.b & A.r * B.g + A.g * B.a \\ A.b * B.r + A.a * B.b & A.b * B.g + A.a * B.a \end{pmatrix}$$

In cg it looks like

```
C.rgba += A.rrbb*B.rgrg + A.ggaa*B.baba
```

2x2 blocking

- Matrix effective size is *N*/2
  - 1/2 the rendering passes
  - 1/8 the memory accesses
  - But each access 4 times larger (float4)
    - 1/2 the bandwidth requirement
- Reminder: 1/4 the GPU flops

4x1 blocking

Moravanszky [2003]:

- Same but blocks are 4x1
- Cannot multiply two 4x1 blocks
  - size mismatch
- Instead, multiply 4x4 block by 4x1
  - Result is another 4x1 block

4x1 blocking

Product of 4x4 matrix and 4x1 matrix:

$$\begin{pmatrix} C.r \\ C.g \\ C.b \\ C.a \end{pmatrix} = \left(\begin{pmatrix} A1.r \\ A1.g \\ A1.b \\ A1.a \end{pmatrix} \begin{pmatrix} A2.r \\ A2.g \\ A2.b \\ A2.a \end{pmatrix} \begin{pmatrix} A3.r \\ A3.g \\ A3.b \\ A3.a \end{pmatrix} \begin{pmatrix} A4.r \\ A4.g \\ A4.b \\ A4.a \end{pmatrix} \right) \times \begin{pmatrix} B.r \\ B.g \\ B.b \\ B.a \end{pmatrix}$$

In Cg:

```
C.rgba += A1.rgba*B.rrrr + A2.rgba*B.gggg
        + A3.rgba*B.bbbb + A4.rgba*B.aaaa
```

4x1 blocking

- Could be 1/4 the rendering passes
  - But 2 times longer fragment shader
  - Likely, have 1/2 the rendering passes,
    - same as in 2x2 case
- 5 memory references per 4 vector multiplies
  - Was 2 references for 2 multiplies in 2x2 blocking
  - Bandwidth-wise less efficient

Multiplying larger blocks

- Consider shaders that
  - Multiply 2x2 blocks
  - Multiply 4x4 matrix by 4x1 vector
- What about multiplying 4x4 blocks?
  - Or larger?
- 4x4 times 4x4 vs. 4x4 times 4x1
  - 3 more texture fetches (8 total)
  - 12 more vector multiplies (16 total)
  - 16/8 vs. 4/5 – much more efficient!

Multiplying 4x4 blocks

$$\left(\begin{pmatrix} C1.r \\ C1.g \\ C1.b \\ C1.a \end{pmatrix} \begin{pmatrix} C2.r \\ C2.g \\ C2.b \\ C2.a \end{pmatrix} \begin{pmatrix} C3.r \\ C3.g \\ C3.b \\ C3.a \end{pmatrix} \begin{pmatrix} C4.r \\ C4.g \\ C4.b \\ C4.a \end{pmatrix} \right) =$$

$$= \left(\begin{pmatrix} A1.r \\ A1.g \\ A1.b \\ A1.a \end{pmatrix} \begin{pmatrix} A2.r \\ A2.g \\ A2.b \\ A2.a \end{pmatrix} \begin{pmatrix} A3.r \\ A3.g \\ A3.b \\ A3.a \end{pmatrix} \begin{pmatrix} A4.r \\ A4.g \\ A4.b \\ A4.a \end{pmatrix} \right) \times \left(\begin{pmatrix} B1.r \\ B1.g \\ B1.b \\ B1.a \end{pmatrix} \begin{pmatrix} B2.r \\ B2.g \\ B2.b \\ B2.a \end{pmatrix} \begin{pmatrix} B3.r \\ B3.g \\ B3.b \\ B3.a \end{pmatrix} \begin{pmatrix} B4.r \\ B4.g \\ B4.b \\ B4.a \end{pmatrix} \right)$$

Multiplying 4x4 blocks

- The product is another 4x4 matrix
  - How to output it from shader?
- GPUs' multiple render targets
- But:
  - Need more registers

Performance Results

Fatahalian et al. 2004:

- Implemented these techniques
- Present benchmarks
- Draw curious conclusions

Multiplication of 1024x1024 matrices

	Time (s)	**GFLOPS**	BW (GB/s)
NV 5900 Single	0.781	**2.75**	7.22
NV 5900 Multi	0.713	**3.01**	9.07
NV 6800 Single	0.283	**7.59**	15.36
NV 6800 Multi	0.469	**4.58**	12.79
ATI 9800 Multi	0.445	**4.83**	12.06
ATI X800 Multi	0.188	**11.4**	27.5
P4 3GHz (ATLAS)	*0.289*	***7.78***	*27.68*

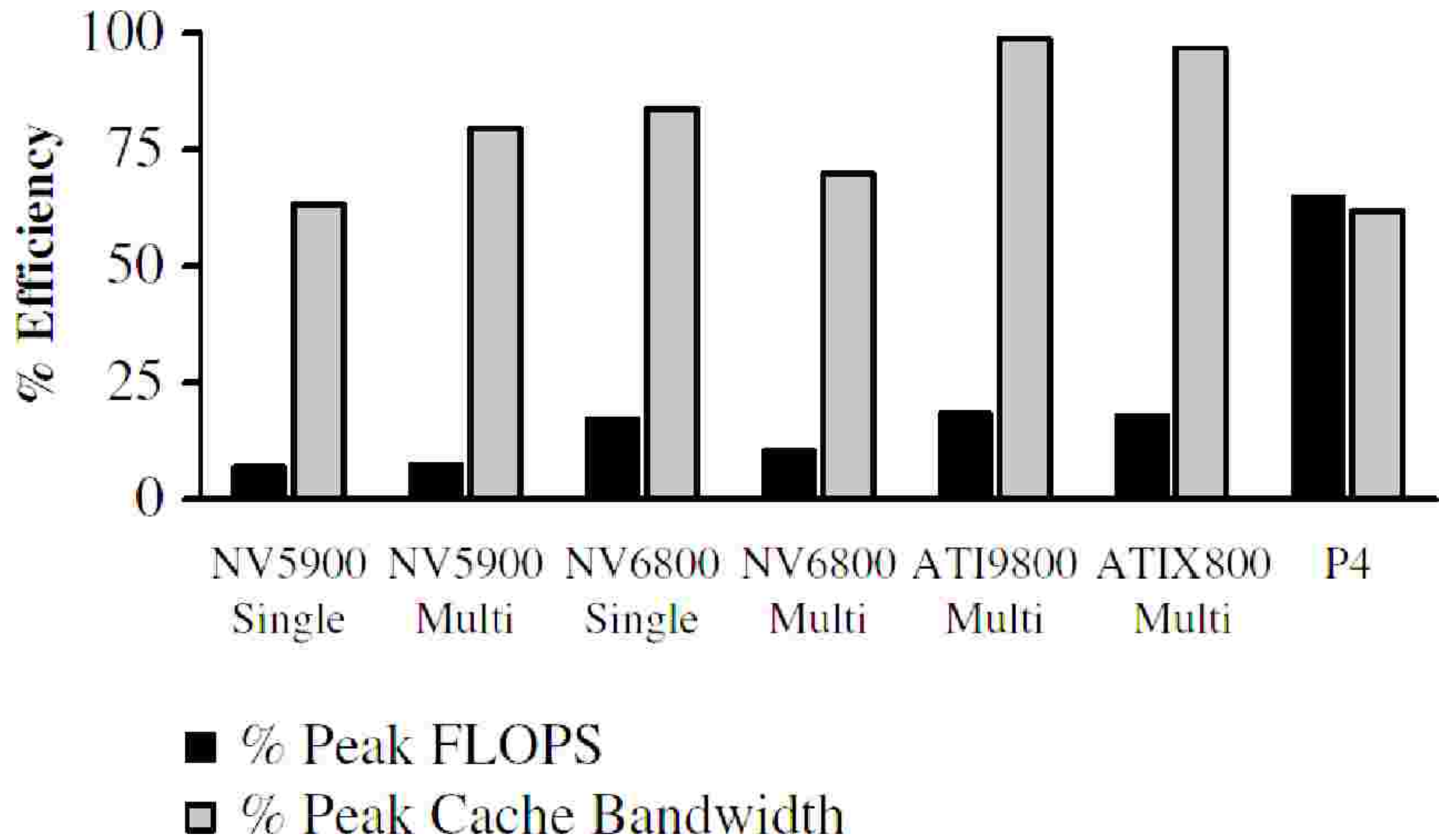
Peak rates

- Measure *peak computation* rate
  - `mad` instructions only
- Measure *peak cache* bandwidth
  - Access texel (0,0) only
- Measure peak memory bandwidth
  - 1-to-1 copy of input texture

Peak Rates

	GFLOPS	Cache BW	Mem BW
NV 5900	40.0	11.4	4.1
NV 6800	43.9	18.3	3.8
ATI 9800	26.1	12.2	7.3
ATI X800	63.7	28.4	15.6
P4 3GHz	*12*	*44.7 (L1)*	

Percentage efficiency when multiplying 1024x1024 matrices: 2x2 vs. 4x1



Techniques tried but failed

- Multiple rendering targets
  - "unsatisfactory results"
- Tiling
  - No impact found
- Other RGBA packing
- Various number of passes

Conclusions on GPU Cache

- 95% of the cache bandwidth is used
  - But computational units are still idle
- Cache bandwidth does not match FLOPS!
  - Unlike the CPU case

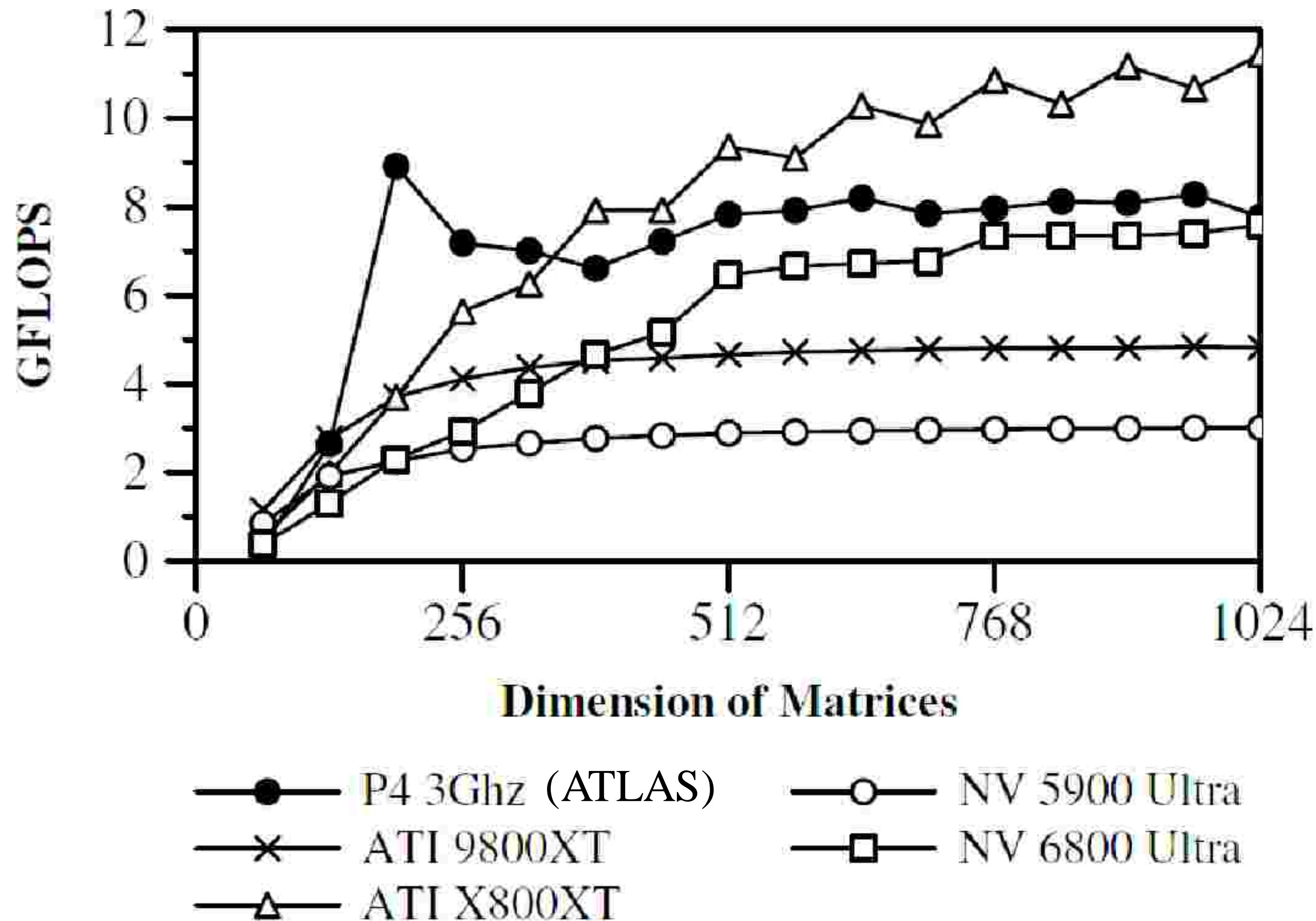
- Sketch of GPU's memory hierarchy:
  - Fast texture memory
  - .... (what about even faster cache?)
  - Registers

Conclusion on Matrix Multiply

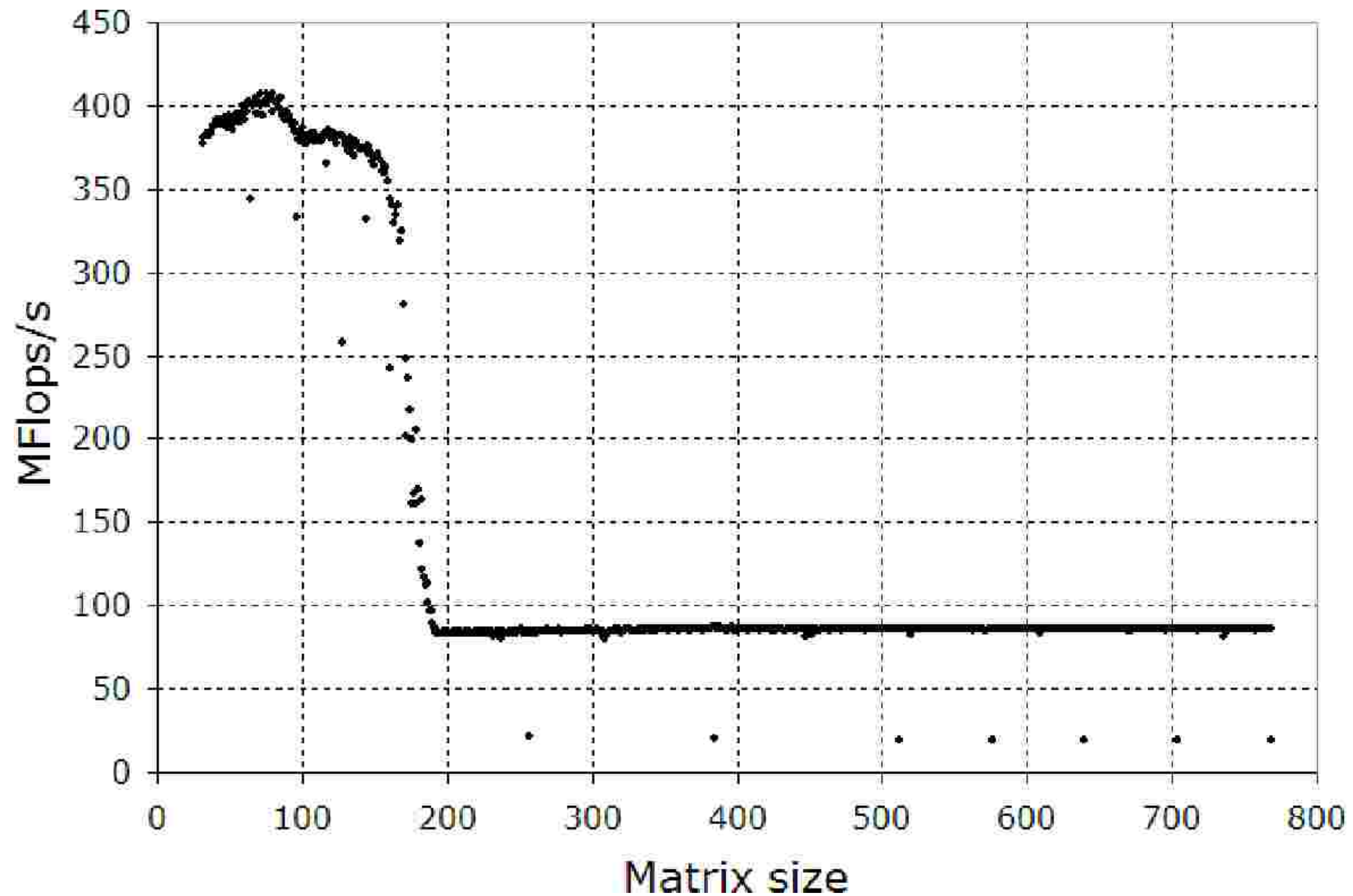
- Need many more flops per memory reference
  - Impossible with present day hardware!

- “No algorithm will perform significantly better than existing approaches”

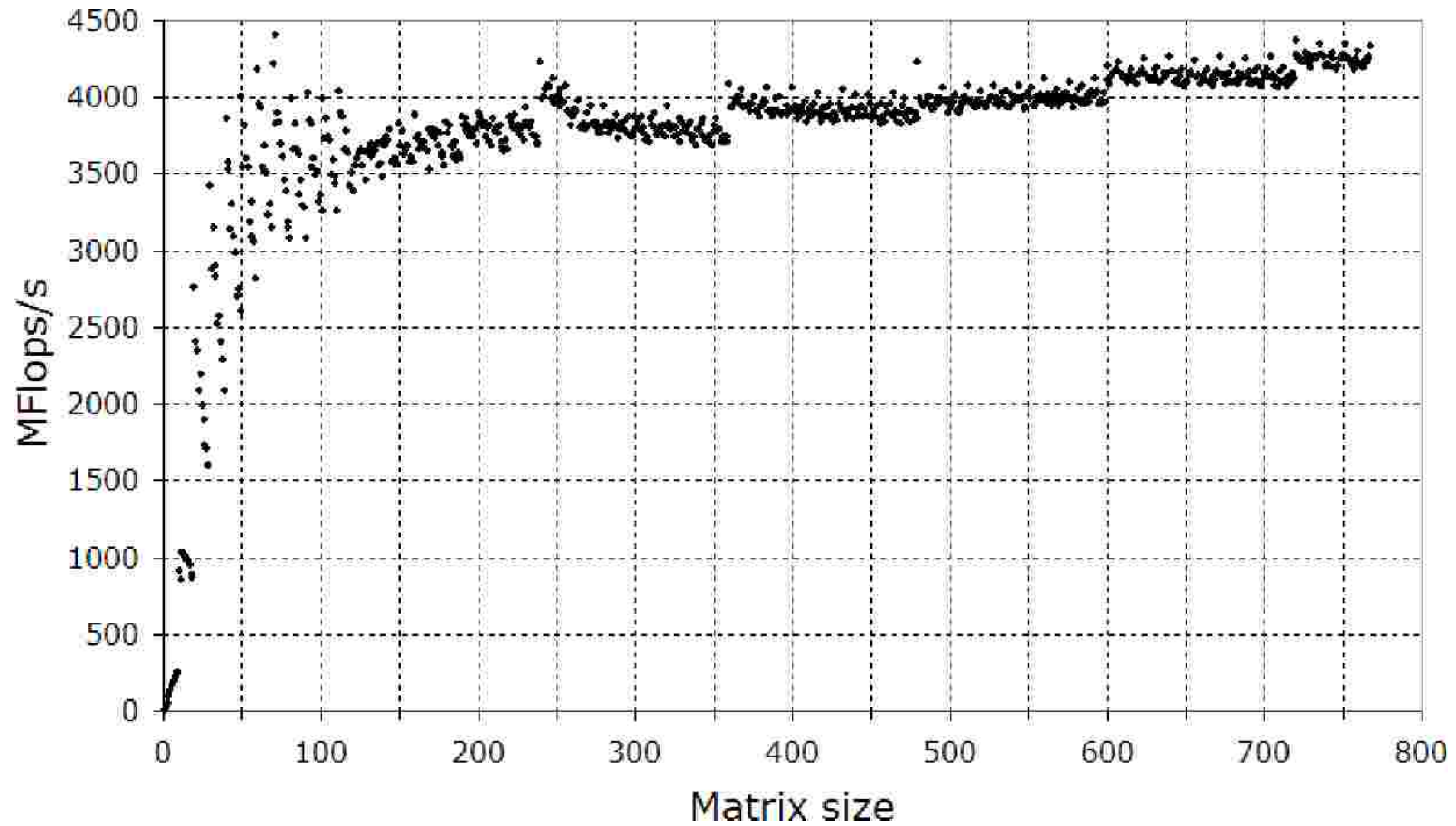
Performance of multiplying square matrices



Comparison: Nonblocked CPU



Vendor-tuned BLAS on CPU



Review of Techniques

- Multichannel blocking: 1x4 and 2x2
- Multiple rendering targets (larger blocking)
- Single pass rendering
- Unrolling (multi-pass)

Which one is better for a specific GPU?

- Try all, find the best (as Fatahalian et al. did)
- Not terribly intellectual work
  – Let computer do it automatically

Automated Tuning on CPU

- Generate parametrized code
- Brute search through all parameter space
- Implemented in:
  - PHiPAC
  - ATLAS
- So good, that ATLAS is used for reference
  - Vendor-tuned libraries are still better

GPU parameter space

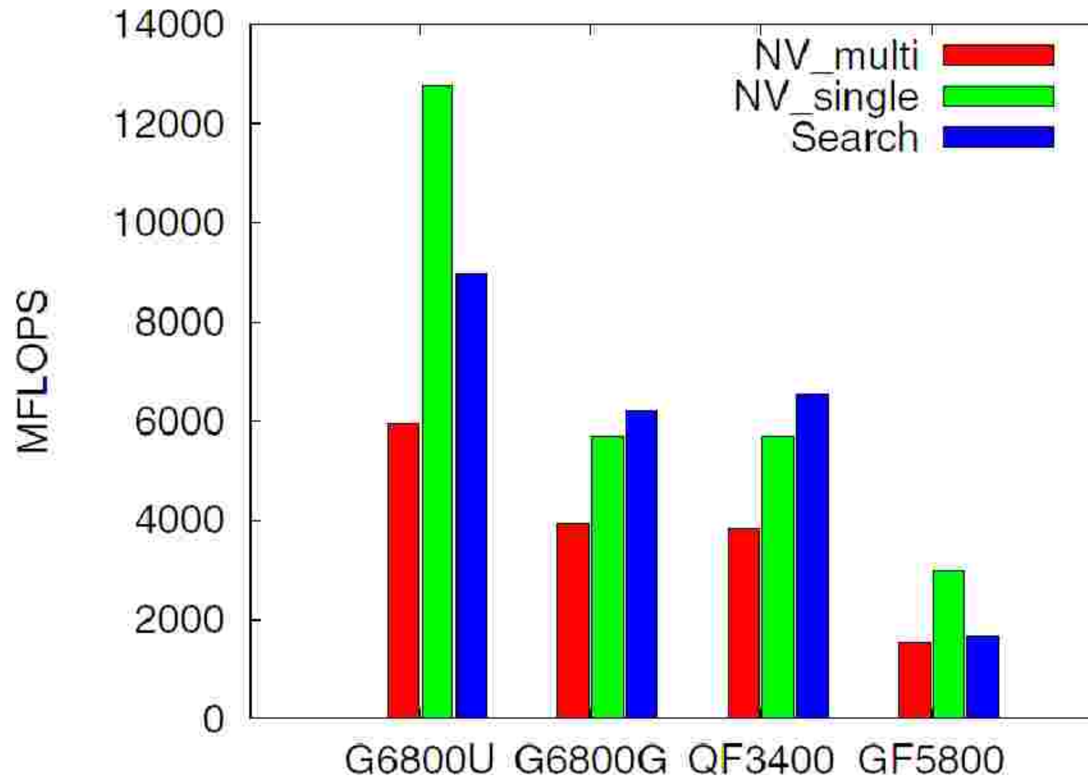
[Jiang and Snir 2005]

- Multiple render target block size
  - 1x1, 1x2, 1x4, 2x2 + symmetric
- RGBA packing
  - 4x1, 2x2, 2x1, 1x1 + symmetric
- Unrolling
  - From 1 to 256 (hardware limit)
- Single pass vs. multi-pass
- Compiler used – cgc, fxc
- Profile used – arb, fp30, fp40, ps20, ps2a, etc.

Parameter space

- $8 \times 8 \times 256 \times 2 \times 2 \times 7 = 458752$ versions
- Oops – 53 days for exhaustive search
- Have to do something
  - Space pruning
    - Assume symmetry in block sizes
    - Search for powers of two values only (for loop unrolling)
    - Assume certain simple dependence on unrolling factor
  - Space decomposition
    - Assume unrolling is independent of blocking
- Now it takes 4 hours only

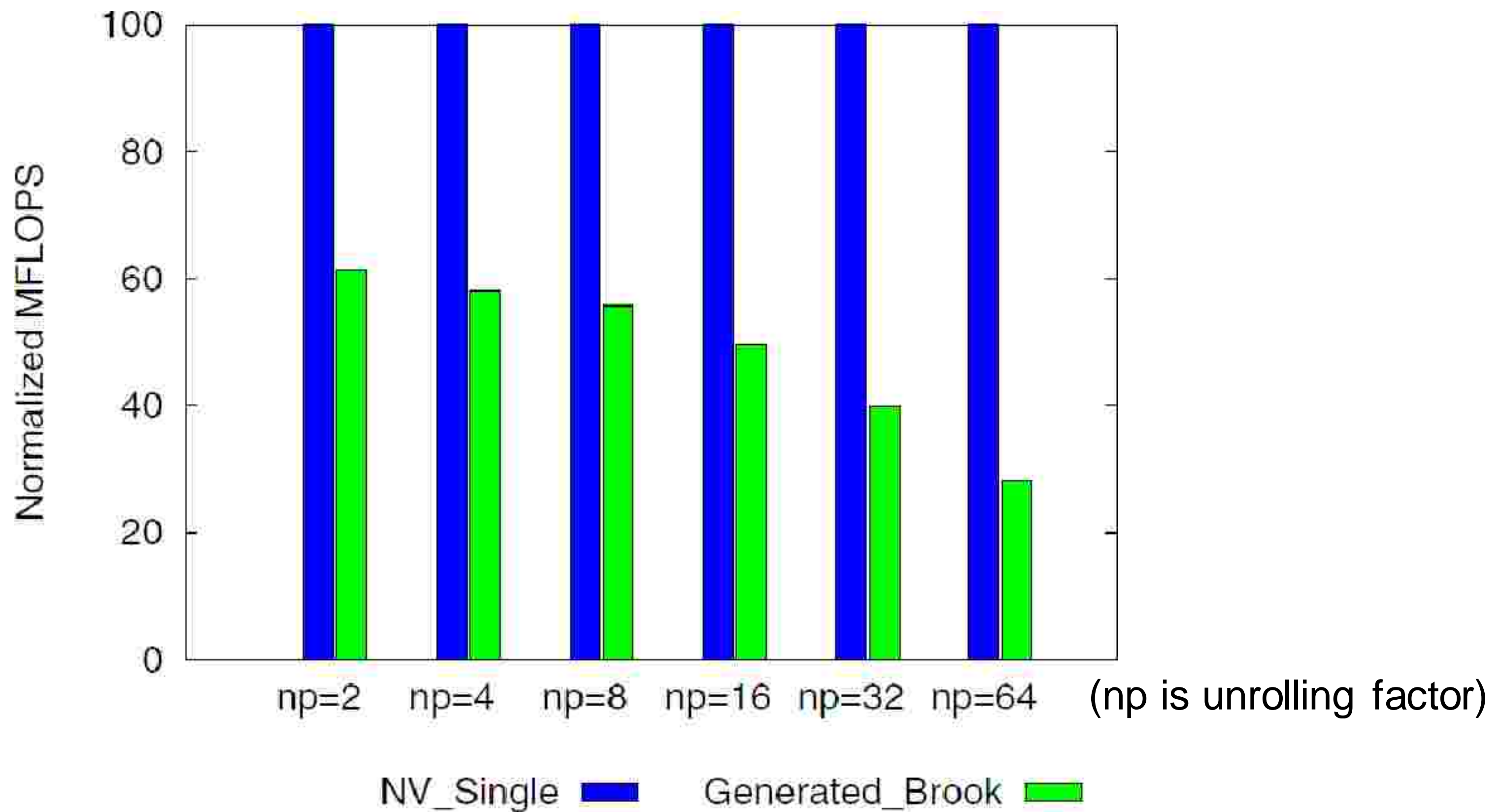
The results



NV_single was tuned for GeForce 6800 Ultra (G6800U)

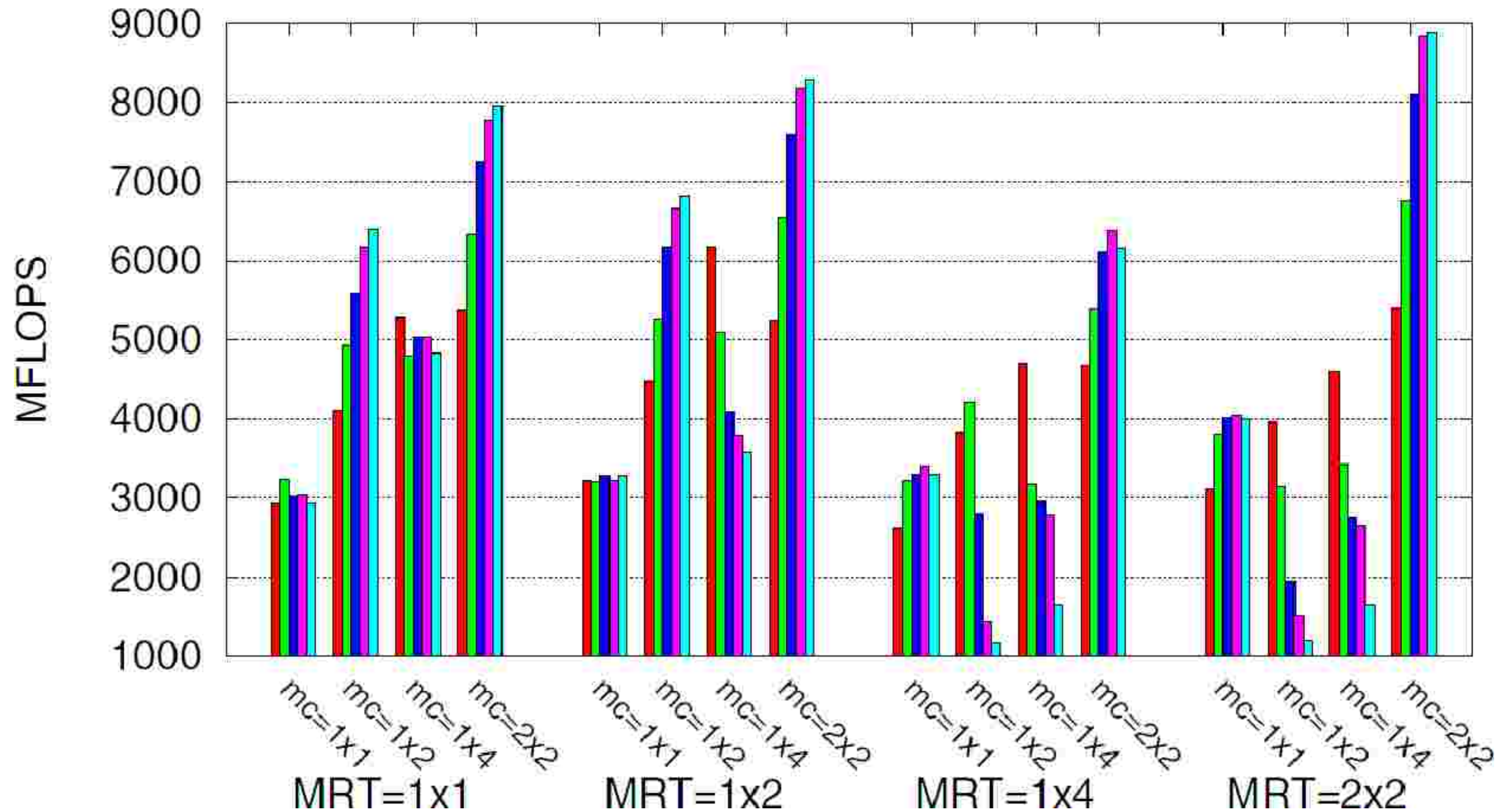
NV_* are due to Fatahalian et al. [2004]

Comparison with hand-tuning



They use BrookGPU and blame it for this performance deficiency

Sensitivity to block sizes



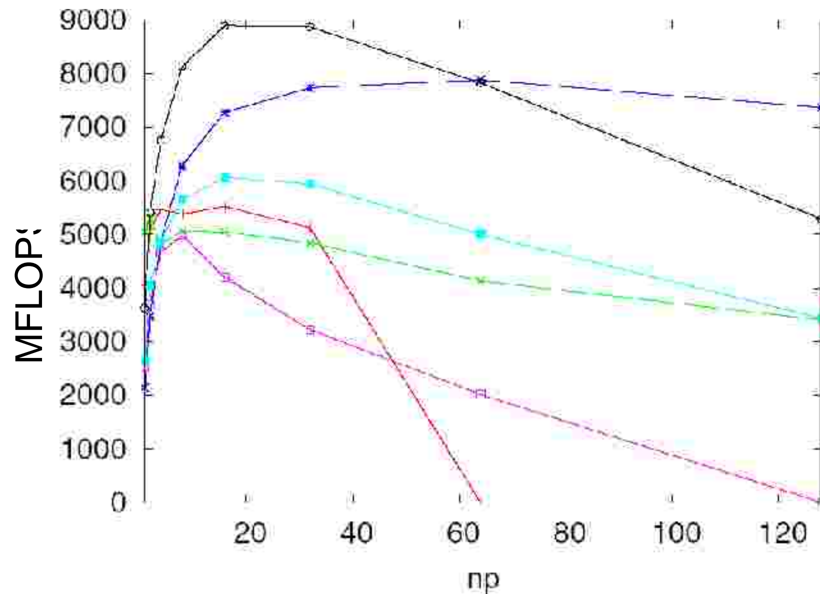
mc – RGBA packing

MRT – multiple rendering targets (1x1 is single target)

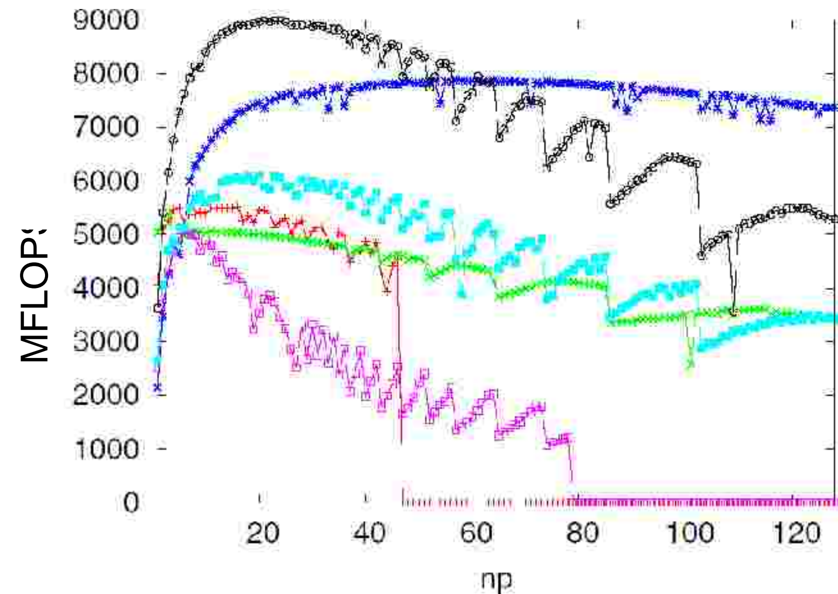
Colors: unroll factor in {2, 4, 8, 16, 32}

Dependence on unrolling factor

Powers of two



All values



(np is unrolling factor)

Can we guess the cache size from the graph?

Other results

- Single pass version was inefficient
  - Compiler splits it in multiple passes anyway
- Better profiles perform better
  - fp40 was better than fp30
  - fp30 was better than arb
- *fxc* and *cgc* generate equivalently fast code

Summary

- If you do bandwidth-bound tasks with large data reuse:
  - Pack data into RGBA
  - Reduce number of rendering passes
  - Reduce # of memory references per flop
  - Use register-level data reuse
- Do not rely on cache-level data reuse
  - Cache is small and slow

References

- E.S. Larsen, and D. McAllister, 2001. Fast matrix multiplies using graphics hardware, In *Supercomputing 2001*.
- Ádám Moravánszky, 2003. Dense matrix algebra on the GPU, in *ShaderX^2 – Shader Tips & Trics*, ch. 2.
- J. Hall, N. Carr, and J. Hart, 2003. Cache and bandwidth aware matrix multiplication on the GPU, Tech. Report, UIUC.
- K. Fatahalian, J. Sugerman, and P. Hanrahan, 2004. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Graphics Hardware 2004*.
- C. Yeo, 2004. Numerical Algorithms on GPU, *MA221 Class Project Writeup.*
- C. Jiang, and M. Snir, 2005. Automatic Tuning Matrix Multiplication Performance on Graphics Hardware.

Acknowledgements

- Vasily Volkov put together the original version of these slides