The GPU as a co-processor in FEM-based simulations

Preliminary results

Dipl.-Inform. Dominik Göddeke

dominik.goeddeke@mathematik.uni-dortmund.de

Institute of Applied Mathematics
University of Dortmund

LS3

Outline

- Motivation and general introduction (Robert)
- Techniques and data layouts
- Performance of basic numerical linear algebra components
- Efficiency vs. accuracy using 16bit floating point arithmetics
- Towards ”numerical GIGAFLOP/s”
- Discussion

Hardware

All tests presented have been implemented in OpenGL + Cg on a Windows box.

Hardware

All tests presented have been implemented in OpenGL + Cg on a Windows box.

Four different systems have been evaluated:

- AMD Opteron (1800 MHz) as CPU reference with SBBLAS benchmark linked to GOTO BLAS (check Christian's talk for details)
- NVIDIA 5950Ultra (NV30, 450 MHz, 4 vertex–, 8 fragment pipelines, 256bit memory interface, 425 MHz GDDR)
- NVIDIA 6800 (NV40, 350 MHz, 5 vertex–, 12 fragment pipelines, 256bit memory interface, 500 MHz GDDR2)
- NVIDIA 6800GT (NV40, 350 MHz, 6 vertex–, 16 fragment pipelines, 256bit memory interface, 500 MHz GDDR2), courtesy of Hendrik Becker

Techniques

All GPU implementations are based on the following techniques. Visit my homepage for detailed tutorials and code examples.

- Render-to-texture and ”ping-ponging” between double-sided offscreen surfaces for fast iteration-type algorithms with data reuse.
- All calculations are performed in the fragment pipeline, the vertex pipeline is used to generate data which is uniformly interpolated by the rasterizer (e.g. array indices).
- Multitexturing and multipass partitioning for maximum efficiency.

Data layouts

Current NVIDIA GPUs support the following three major render target formats:

Data layouts

Current NVIDIA GPUs support the following three major render target formats:

- one 32 bit floating point value (LUMINANCE, s23e8 IEEE-like, memory imprint 32 bits), used to store a single vector
- four 32 bit floating point values (RGBA32, s23e8, memory imprint 128 bits), used to ”solve four systems simultaneously” (no dependencies between different channels, but different data in each channel).
- four 16 bit floating point values (RGBA16, s10e5, memory imprint 64 bits), again to ”solve four systems simultaneously”.

Remark: ATI only supports one to four 24 bit channels.

Numerical linear algebra (I)

The following low-level building blocks for FEM codes have been mapped to the GPU:

- BLAS `SAXPY_C`: $2N$ flops: $\mathbf{y}_i = \mathbf{y}_i + \alpha \mathbf{x}_i, \quad i = 1 \ldots N$
- `SAXPY_V`: $2N$ flops: $\mathbf{y}_i = \mathbf{y}_i + \mathbf{a}_i \mathbf{x}_i, \quad i = 1 \ldots N$
- `MV_V` for a 9-banded FEM (Q_1) matrix with variable coefficients: $18N$ flops, implemented as a series of 9 `SAXPY_V` operations with appropriate zero padding: $\mathbf{y} = \mathbf{y} + A\mathbf{x}$
- `DOT`: $2N$ flops, implemented as a logarithmic reduction: $y = \sum_{i=1}^{N} \mathbf{a}_i \mathbf{b}_i$
- `NORM`: $2N$ flops, implemented as a logarithmic reduction: $y = \sqrt{\sum_{i=1}^{N} \mathbf{a}_i \mathbf{a}_i}$

LS3

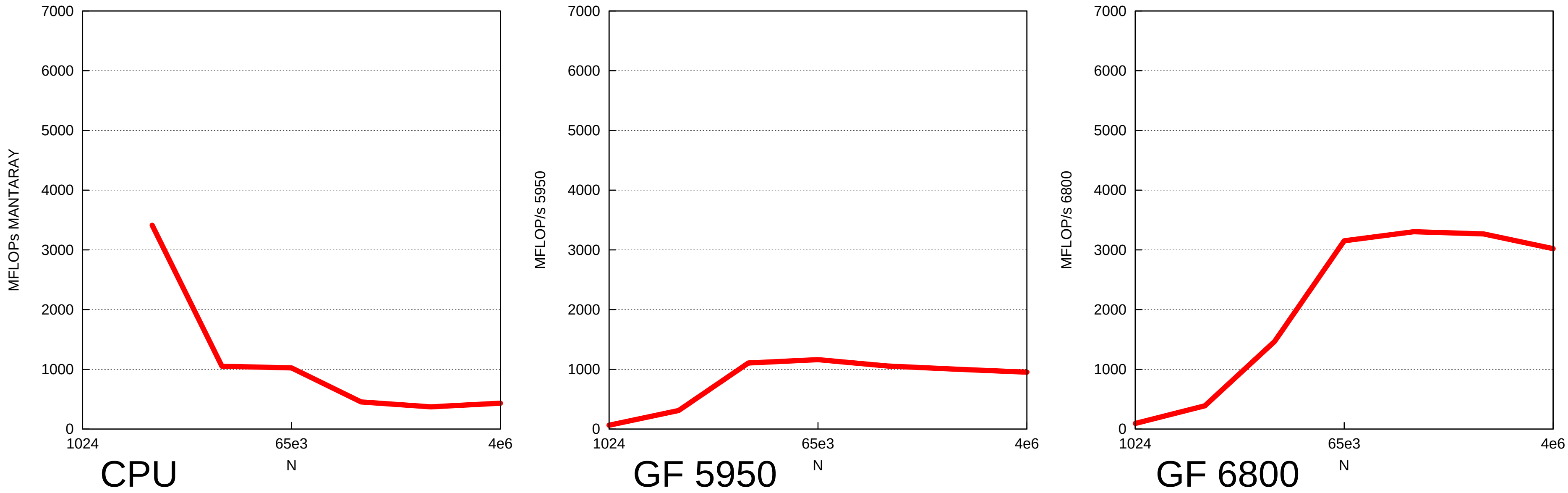
Numerical linear algebra (II)

MFLOP/s rates for LUMINANCE data structure:

LS3

Numerical linear algebra (II)

MFLOP/s rates for LUMINANCE data structure:

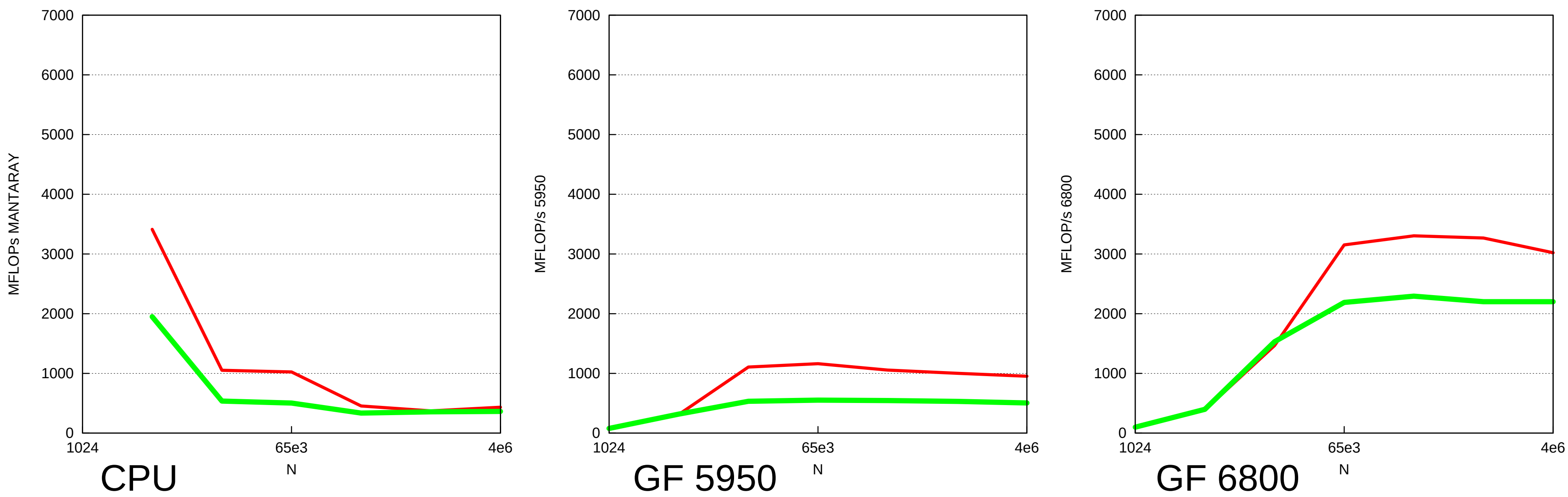


SAXPY_C

LS3

Numerical linear algebra (II)

MFLOP/s rates for LUMINANCE data structure:

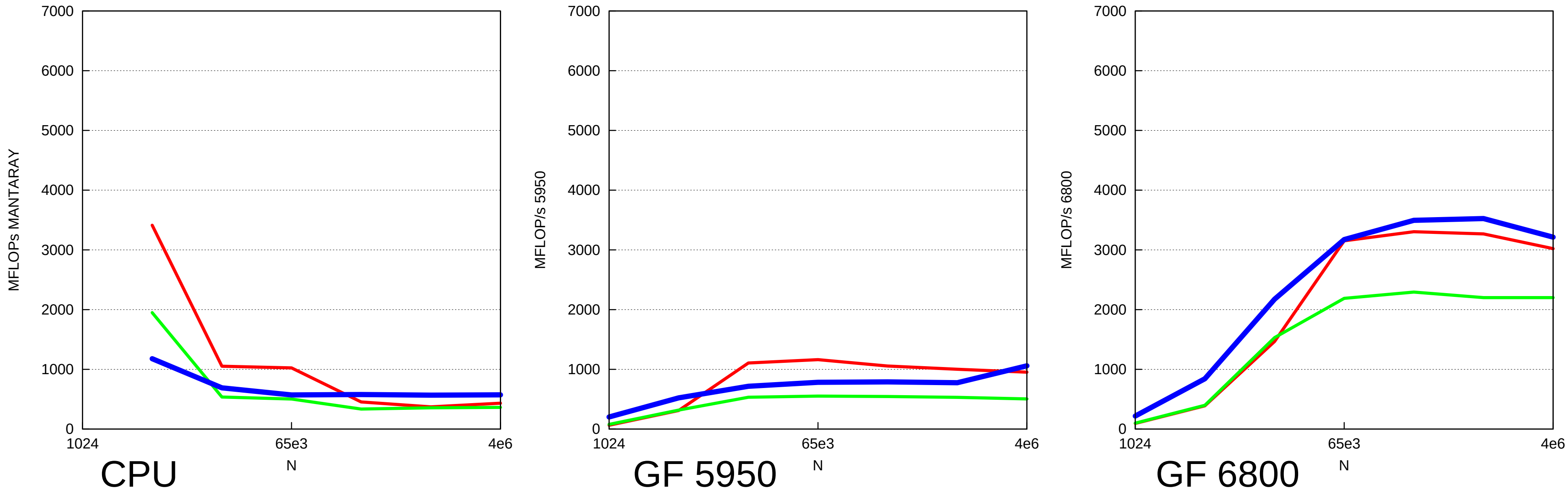


SAXPY_C SAXPY_V

LS3

Numerical linear algebra (II)

MFLOP/s rates for LUMINANCE data structure:

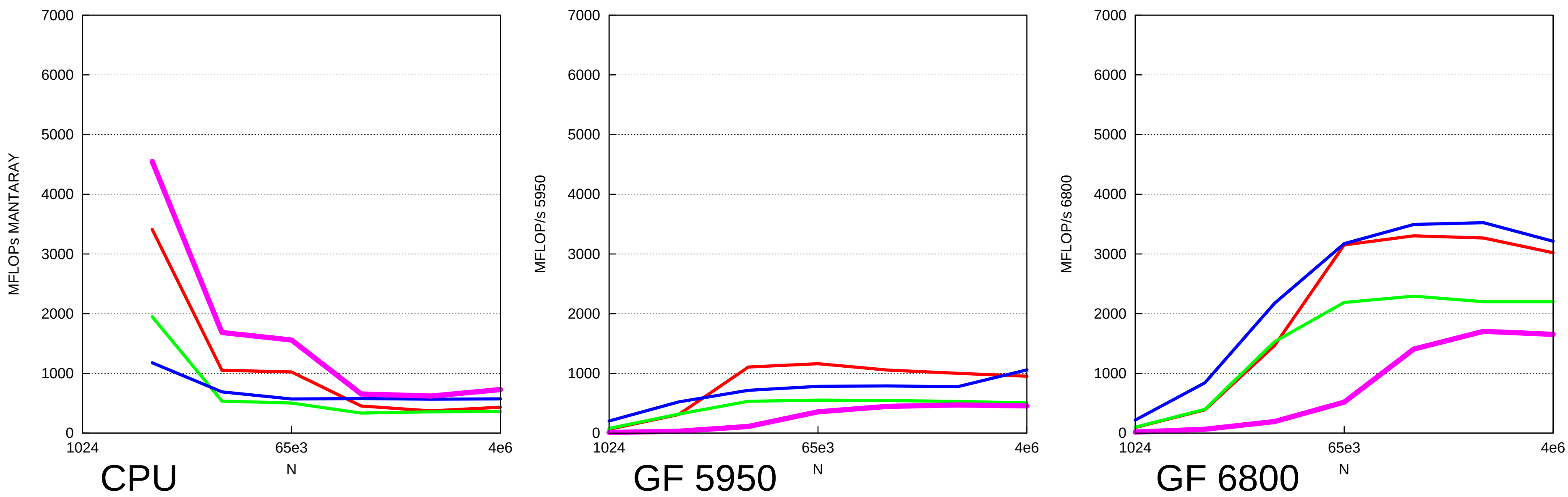


SAXPY_C SAXPY_V MV_V

LS3

Numerical linear algebra (II)

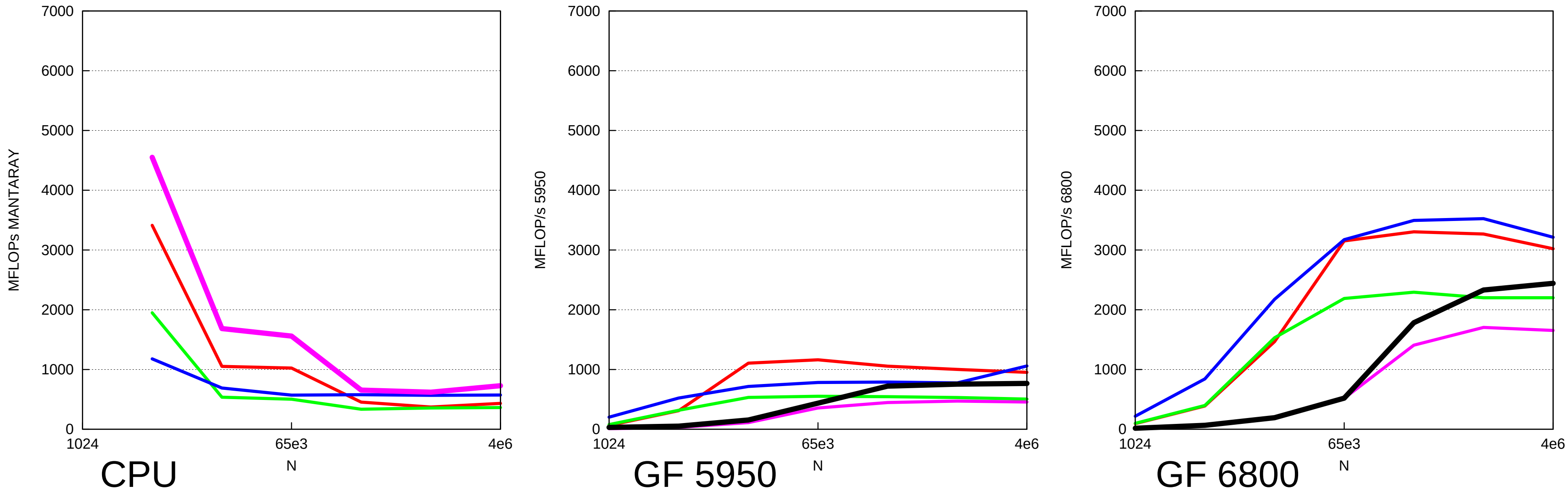
MFLOP/s rates for LUMINANCE data structure:



SAXPY_C SAXPY_V MV_V DOT

Numerical linear algebra (II)

MFLOP/s rates for LUMINANCE data structure:



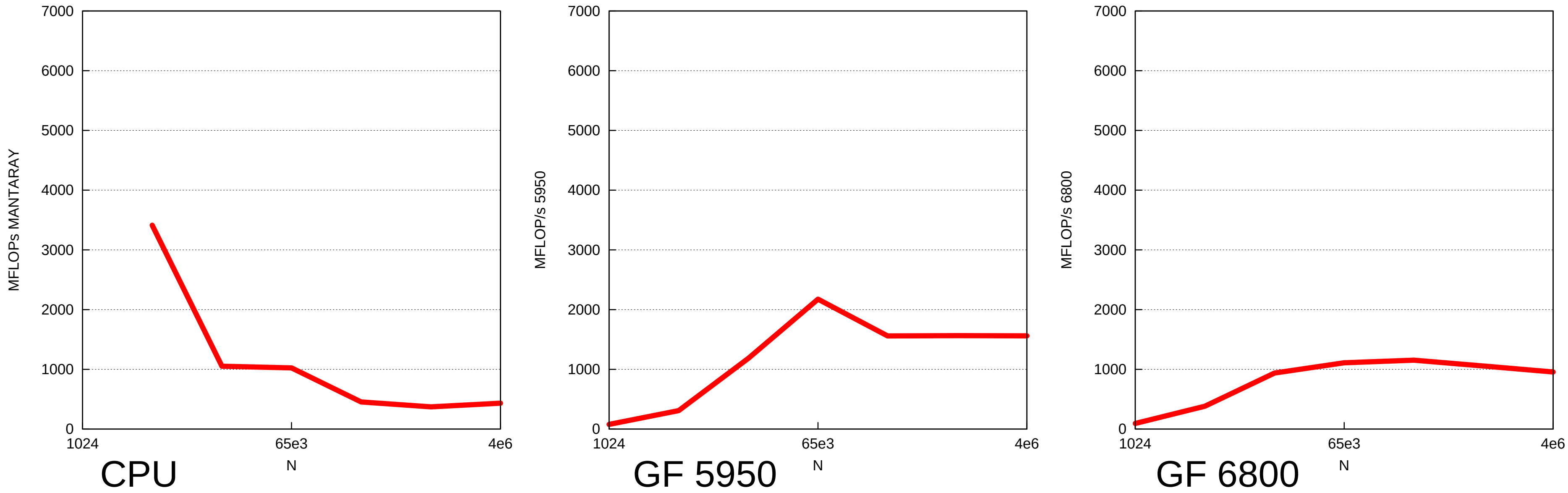
SAXPY_C SAXPY_V MV_V DOT NORM

Numerical linear algebra (III)

MFLOP/s rates for RGBA32 data structure:

Numerical linear algebra (III)

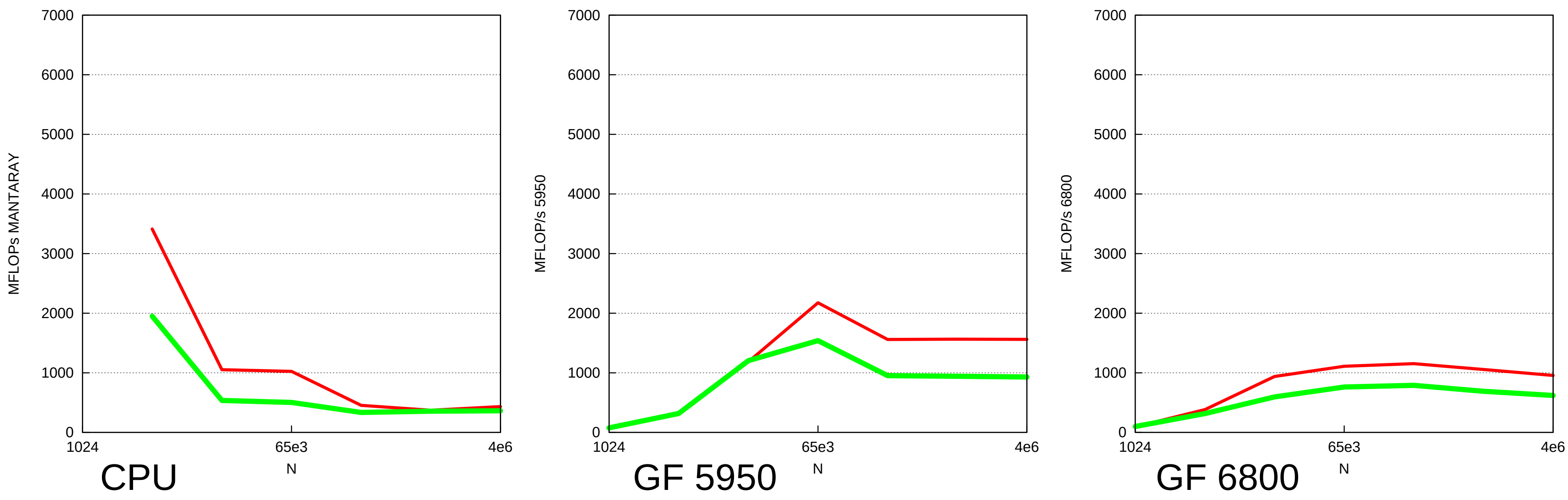
MFLOP/s rates for RGBA32 data structure:



SAXPY_C

Numerical linear algebra (III)

MFLOP/s rates for RGBA32 data structure:

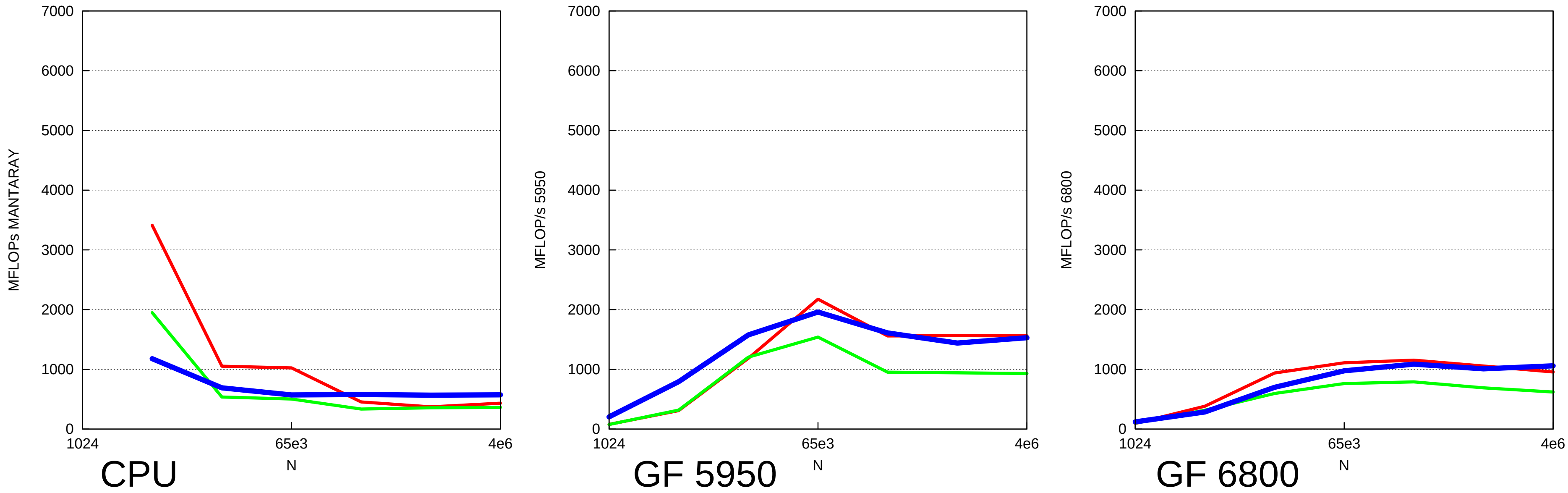


SAXPY_C SAXPY_V

LS3

Numerical linear algebra (III)

MFLOP/s rates for RGBA32 data structure:

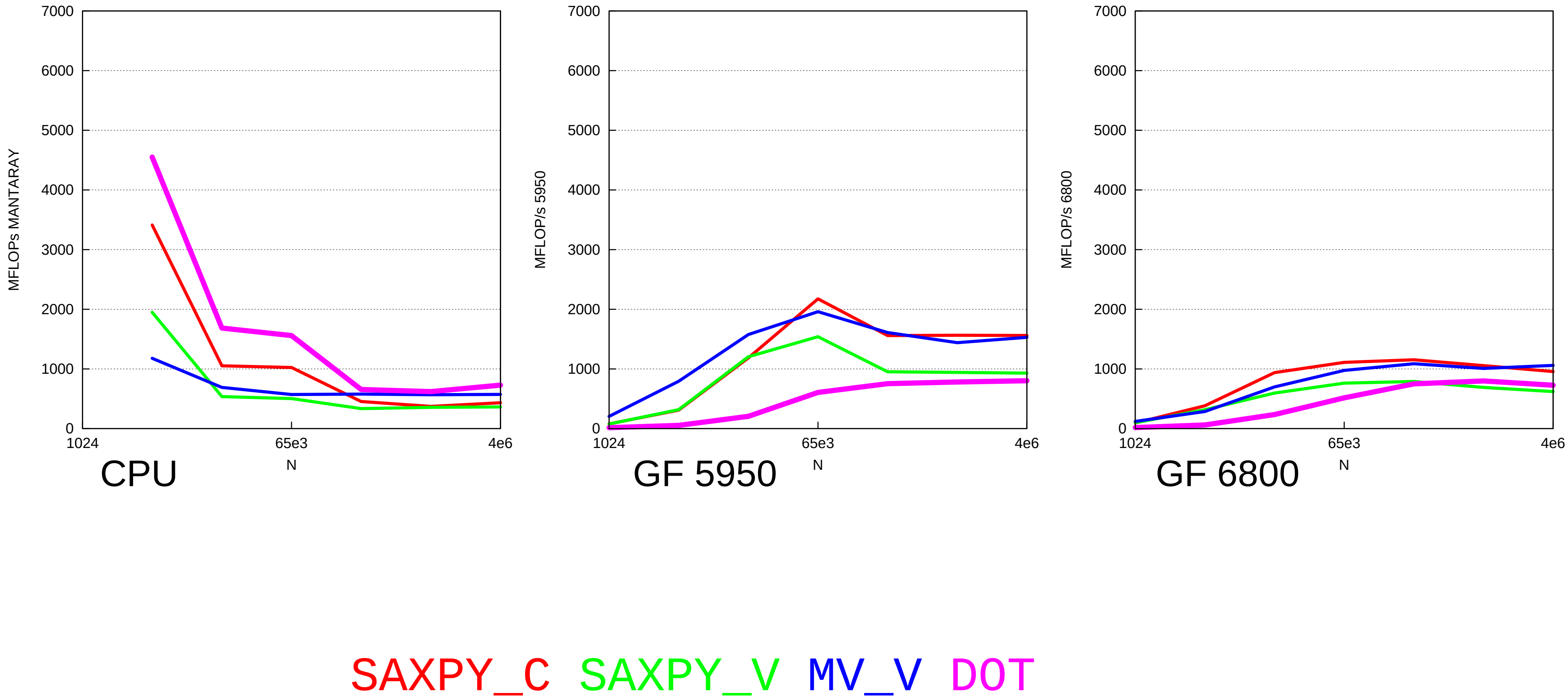


SAXPY_C SAXPY_V MV_V

LS3

Numerical linear algebra (III)

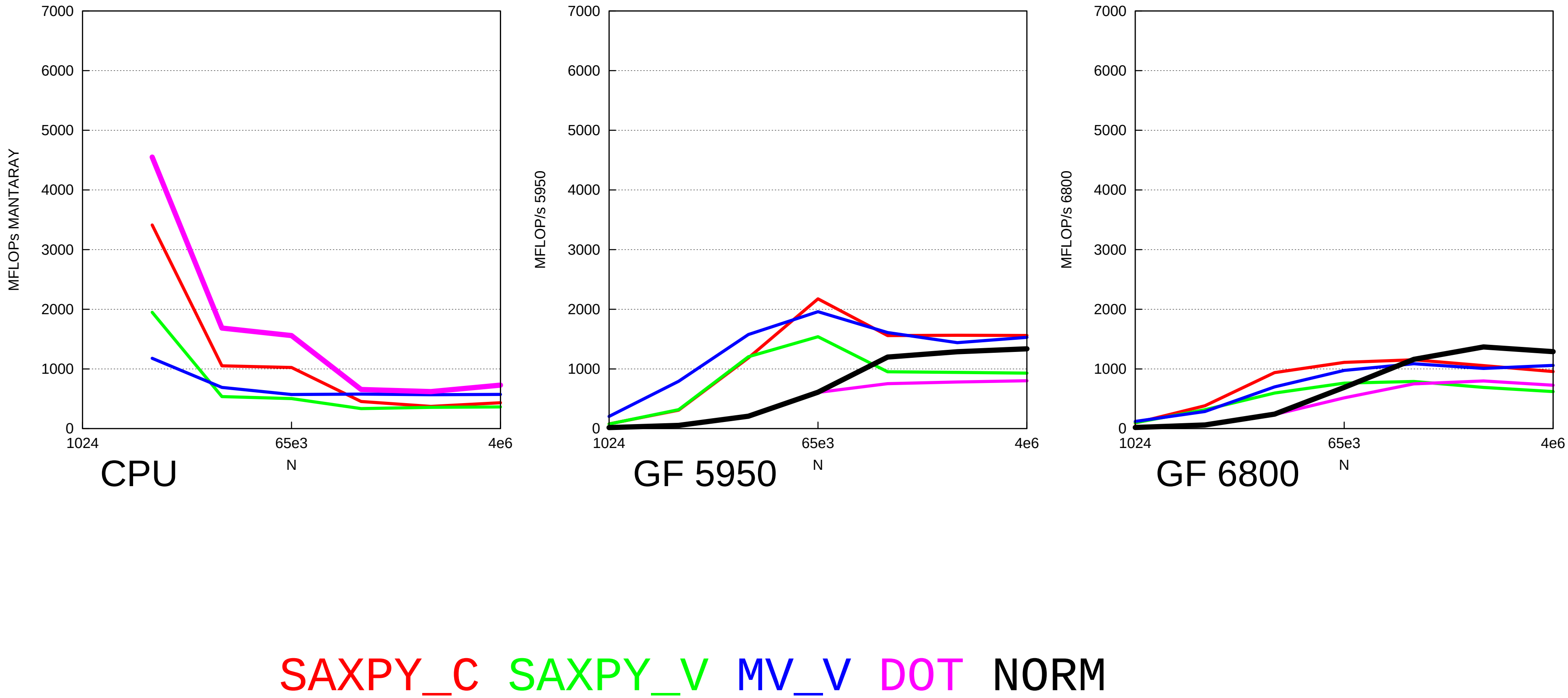
MFLOP/s rates for RGBA32 data structure:



LS3

Numerical linear algebra (III)

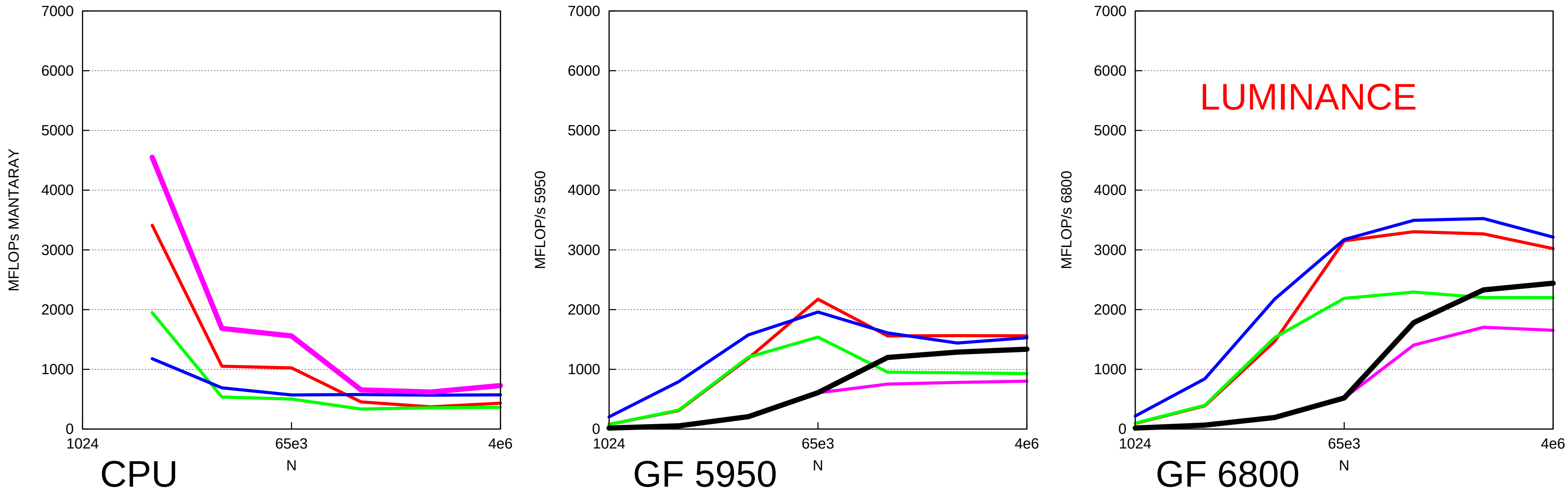
MFLOP/s rates for RGBA32 data structure:



LS3

Numerical linear algebra (III)

MFLOP/s rates for RGBA32 data structure:



SAXPY_C SAXPY_V MV_V DOT NORM

LS3

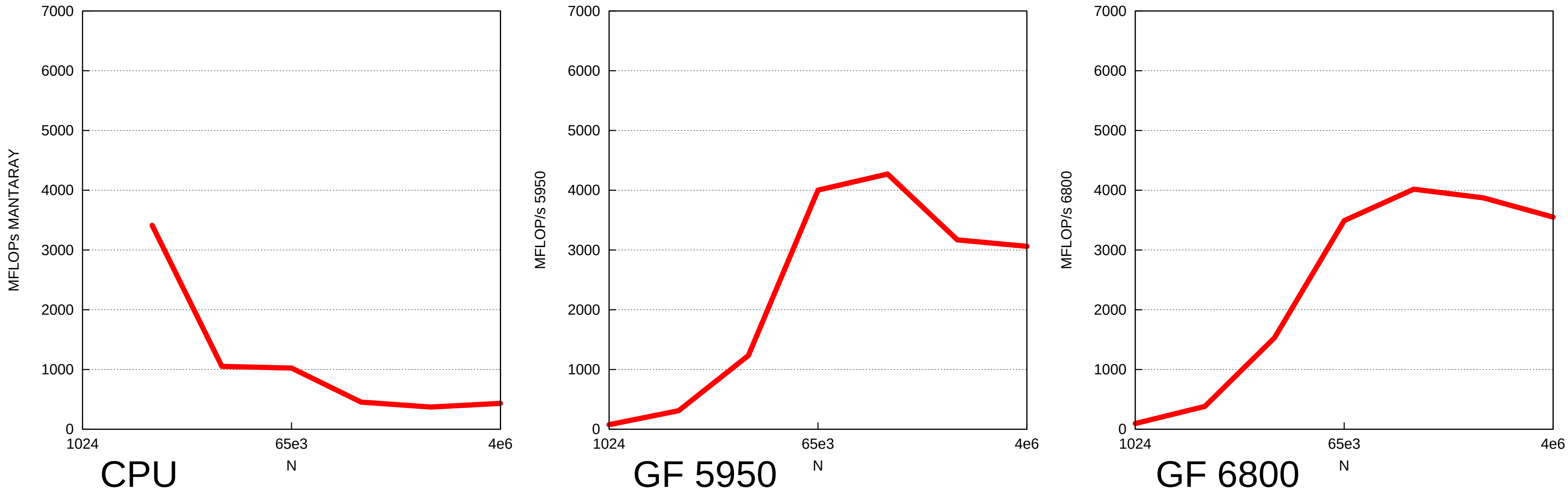
Numerical linear algebra (IV)

MFLOP/s rates for RGBA16 data structure:

LS3

Numerical linear algebra (IV)

MFLOP/s rates for RGBA16 data structure:

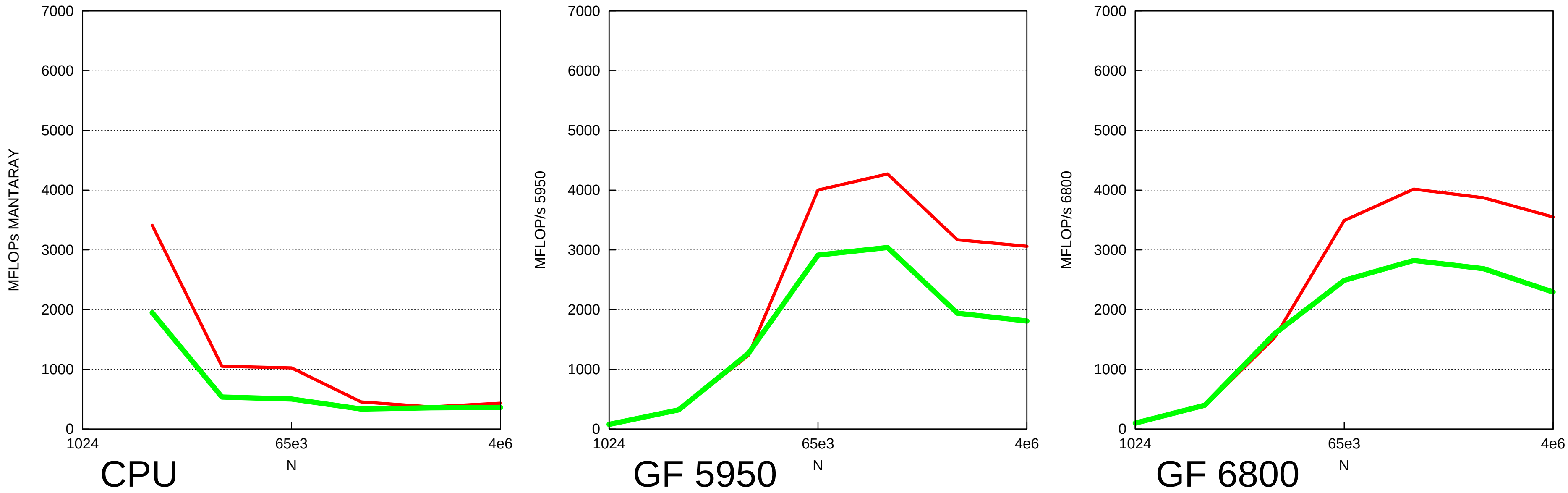


SAXPY_C

LS3

Numerical linear algebra (IV)

MFLOP/s rates for RGBA16 data structure:

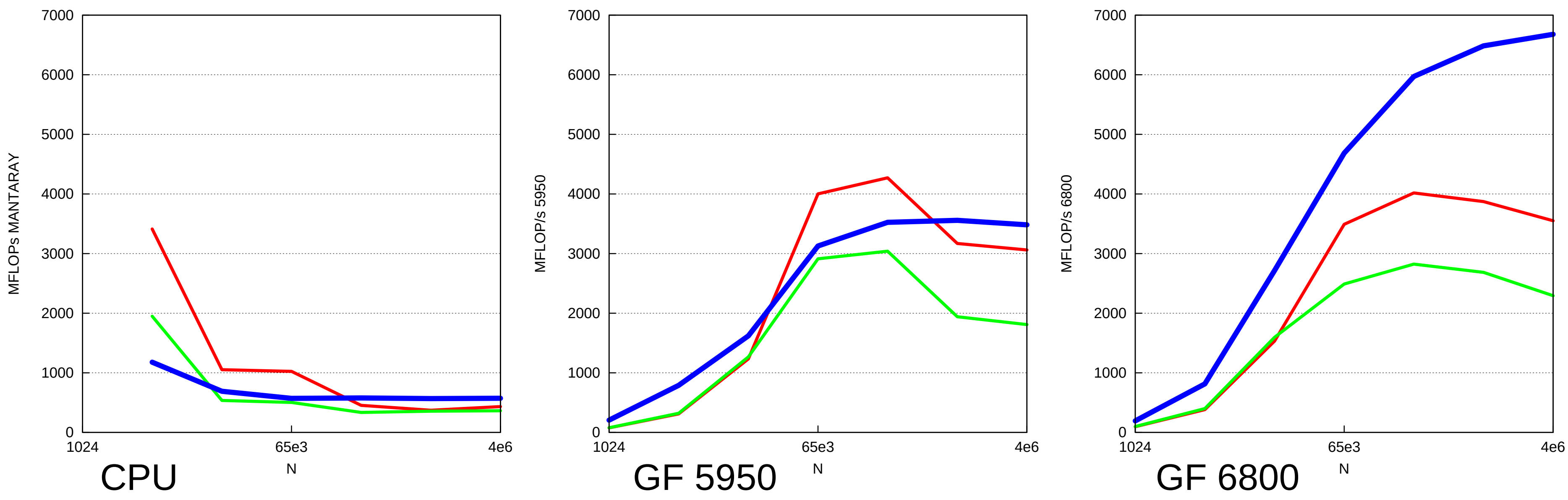


SAXPY_C SAXPY_V

LS3

Numerical linear algebra (IV)

MFLOP/s rates for RGBA16 data structure:

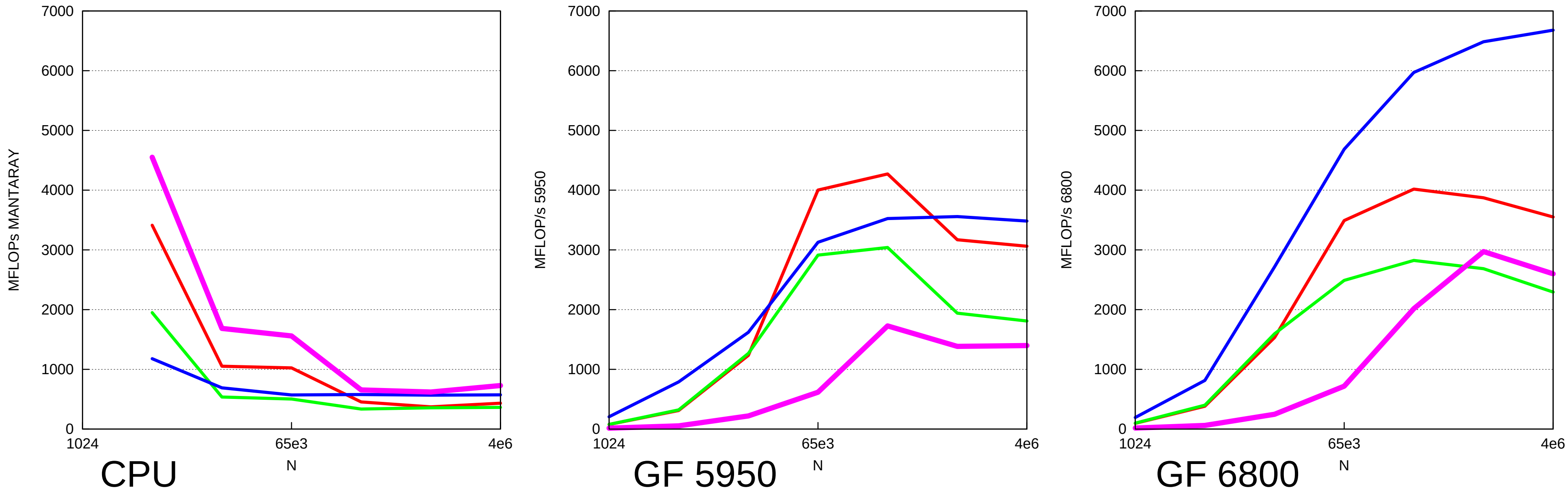


SAXPY_C SAXPY_V MV_V

LS3

Numerical linear algebra (IV)

MFLOP/s rates for RGBA16 data structure:

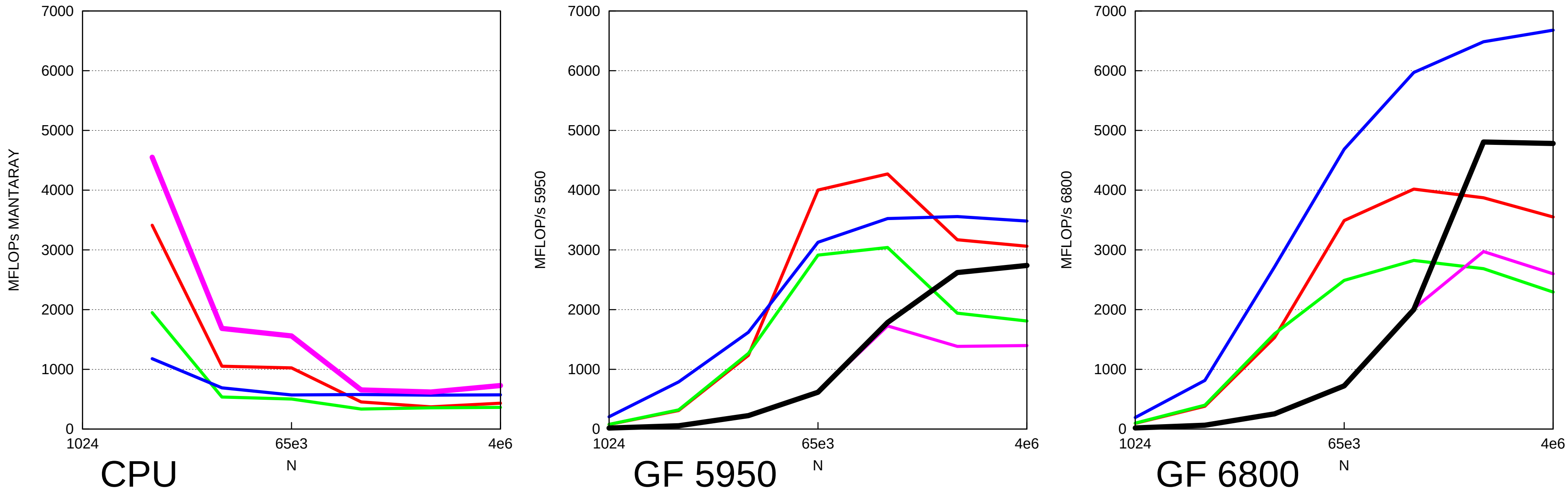


SAXPY_C SAXPY_V MV_V DOT

LS3

Numerical linear algebra (IV)

MFLOP/s rates for RGBA16 data structure:



SAXPY_C SAXPY_V MV_V DOT NORM

LS3

Conclusions and Questions

Conclusions:

- GPUs outperform recent CPUs up to a factor of 5 for single precision arithmetics.
- GPUs only show their true potential for interesting problem sizes that crash the CPU cache.
- Different GPUs behave differently for solving single and quadruple tasks. Appropriate data layouts must be chosen independently for each GPU.
- GPU performance doubles to quadruples for 16 bit floating point arithmetics compared to 32 bit arithmetics.

Conclusions and Questions

Conclusions:

- GPUs outperform recent CPUs up to a factor of 5 for single precision arithmetics.
- GPUs only show their true potential for interesting problem sizes that crash the CPU cache.
- Different GPUs behave differently for solving single and quadruple tasks. Appropriate data layouts must be chosen independently for each GPU.
- GPU performance doubles to quadruples for 16 bit floating point arithmetics compared to 32 bit arithmetics.

Questions:

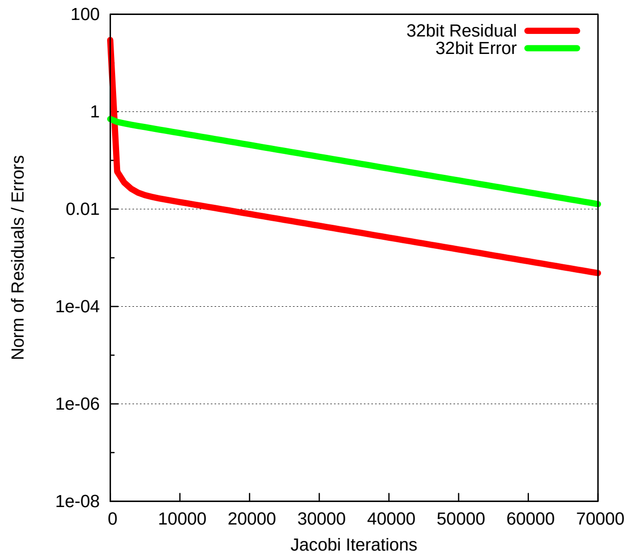
- How can the 16bit performance be achieved while maintaining 32bit accuracy?
- What about the 40 GFLOP/s that were announced?

Accuracy issues with 16 bits

Test case: Solve $A\mathbf{x} = \mathbf{0}$ with 9-band-stencil matrix A and random input $\mathbf{x}$. Use Jacobi scheme based on `MV_V` operator (as prototype for preconditioner and smoother in Krylov space methods) and RGBA (”four independent systems simultaneously”).

Accuracy issues with 16 bits

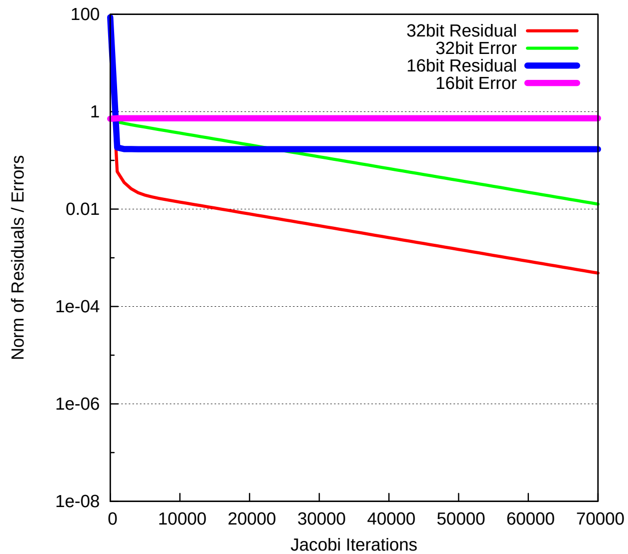
Test case: Solve $A\mathbf{x} = \mathbf{0}$ with 9-band-stencil matrix A and random input $\mathbf{x}$. Use Jacobi scheme based on `MV_V` operator (as prototype for preconditioner and smoother in Krylov space methods) and RGBA (”four independent systems simultaneously”).



LS3

Accuracy issues with 16 bits

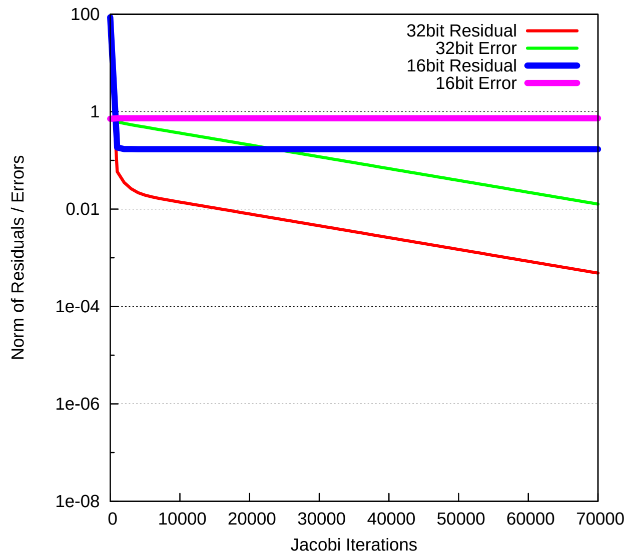
Test case: Solve $A\mathbf{x} = \mathbf{0}$ with 9-band-stencil matrix A and random input $\mathbf{x}$. Use Jacobi scheme based on `MV_V` operator (as prototype for preconditioner and smoother in Krylov space methods) and RGBA (”four independent systems simultaneously”).



LS3

Accuracy issues with 16 bits

Test case: Solve $A\mathbf{x} = \mathbf{0}$ with 9-band-stencil matrix A and random input $\mathbf{x}$. Use Jacobi scheme based on `MV_V` operator (as prototype for preconditioner and smoother in Krylov space methods) and RGBA (”four independent systems simultaneously”).



”Half precision” floats are insufficient for applications beyond visual accuracy. But: Gaining at least one or two decimals is possible, making the use as preconditioner feasible!

LS3

Proof of concept

Use fast 16bit processing as preconditioner, update result ”occasionally” with single or double precision. All GPUs tested so far show identical floating point accuracy.

Proof of concept

Use fast 16bit processing as preconditioner, update result ”occasionally” with single or double precision. All GPUs tested so far show identical floating point accuracy.

1. Calculate defect:
   $\mathbf{d}^{(32)} = A^{(32)}\mathbf{x}^{(32)} - \mathbf{b}^{(32)},\ \alpha = ||\mathbf{d}^{(32)}||$.
2. Check some convergence criterion.

4. Shift solution: $\mathbf{b}^{(16)} = \mathbf{d}^{(32)}$, $\mathbf{x}^{(16)} = \mathbf{0}$.
5. Perform m Jacobi steps to ”solve” $A^{(16)}\mathbf{x}^{(16)} = \mathbf{b}^{(16)}$.
6. Shift corrected solution back:
   $\mathbf{x}^{(32)} = \mathbf{x}^{(32)} - \qquad \mathbf{x}^{(16)}$.

LS3

Proof of concept

Use fast 16bit processing as preconditioner, update result ”occasionally” with single or double precision. All GPUs tested so far show identical floating point accuracy.

1. Calculate defect:
   $\mathbf{d}^{(32)} = A^{(32)}\mathbf{x}^{(32)} - \mathbf{b}^{(32)},\ \alpha = ||\mathbf{d}^{(32)}||.$ CPU or GPU
2. Check some convergence criterion.
   CPU

4. Shift solution: $\mathbf{b}^{(16)} = \mathbf{d}^{(32)}$, $\mathbf{x}^{(16)} = \mathbf{0}$.
   GPU or AGP transfer
5. Perform m Jacobi steps to ”solve” $A^{(16)}\mathbf{x}^{(16)} = \mathbf{b}^{(16)}$.
   GPU
6. Shift corrected solution back:
   $\mathbf{x}^{(32)} = \mathbf{x}^{(32)} - \qquad \mathbf{x}^{(16)}$.
   GPU or AGP transfer

LS3

Proof of concept

Use fast 16bit processing as preconditioner, update result "occasionally" with single or double precision. All GPUs tested so far show identical floating point accuracy.

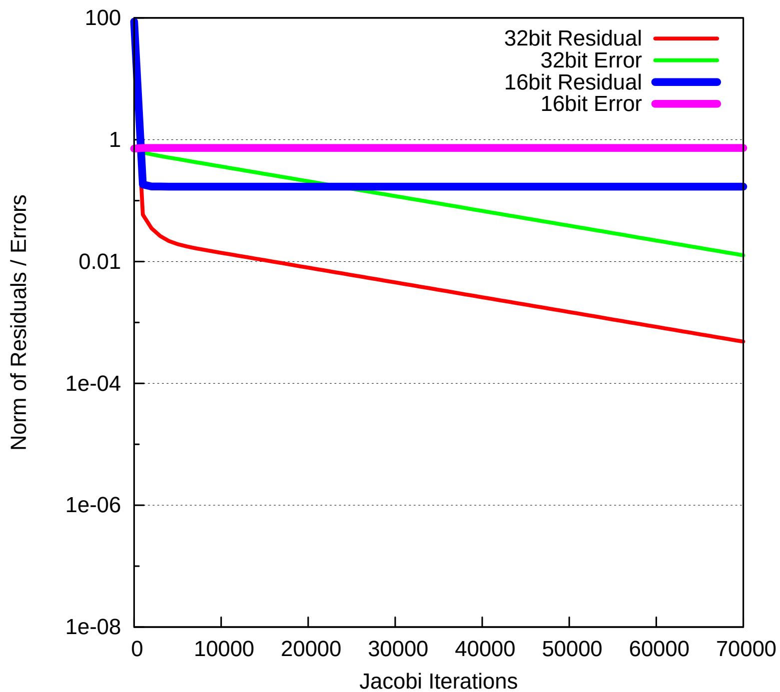
1. Calculate defect: $\mathbf{d}^{(32)} = A^{(32)}\mathbf{x}^{(32)} - \mathbf{b}^{(32)},\ \alpha = ||\mathbf{d}^{(32)}||$.
2. Check some convergence criterion.
3. Apply scaling by defect: $\mathbf{d}^{(32)} = 1/\alpha * \mathbf{d}^{(32)}$.
4. Shift solution: $\mathbf{b}^{(16)} = \mathbf{d}^{(32)}$, $\mathbf{x}^{(16)} = \mathbf{0}$.
5. Perform m Jacobi steps to "solve" $A^{(16)}\mathbf{x}^{(16)} = \mathbf{b}^{(16)}$.
6. Shift corrected solution back: $\mathbf{x}^{(32)} = \mathbf{x}^{(32)} - \omega * \alpha * \mathbf{x}^{(16)}$.

Apply damping by ω and scaling by norm of defect for better convergence and to keep well within the dynamic range of the 16bit "half precision" data type.

LS3

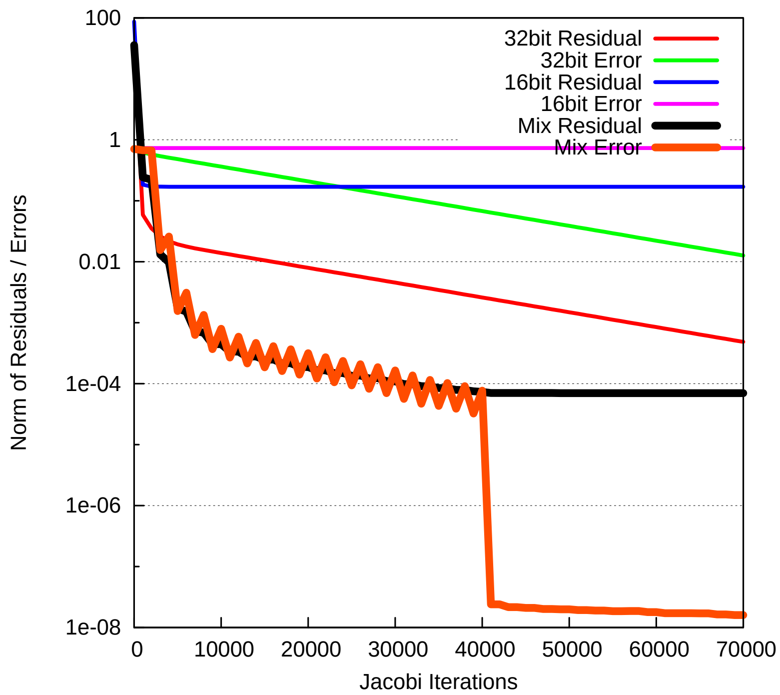
Proof of concept

Results:



Proof of concept

Results:



Proof of concept

Results:

- 32 bit Jacobi iteration: ~ 1100 MFLOPs, 70K iterations
- 16 bit Jacobi iteration: ~ 3800 MFLOPs, ∞ iterations
- Norm: ~ 2000 MFLOPs
- Combined scheme with correction on CPU: $\sim 800-1200$ MFLOPs (depending on problem size), 40K iterations
- Combined scheme running completely on GPU: ~ 3500 MFLOPs (independent of problem size), 40K iterations

LS3

Proof of concept

Results:

- 32 bit Jacobi iteration: ~ 1100 MFLOPs, 70K iterations
- 16 bit Jacobi iteration: ~ 3800 MFLOPs, ∞ iterations
- Norm: ~ 2000 MFLOPs
- Combined scheme with correction on CPU: $\sim 800 - 1200$ MFLOPs (depending on problem size), 40K iterations
- Combined scheme running completely on GPU: ~ 3500 MFLOPs (independent of problem size), 40K iterations

Questions:

- When should the update be performed?
- Can this be predicted a priori to avoid heavy data transfer to CPU?

How to get closer to peak performance?

"Data moving is expensive, not data processing" is valid for GPUs as well! On GPUs, this can be quantified with the arithmetic intensity (number of flops per texture lookup) of implementations.

How to get closer to peak performance?

"Data moving is expensive, not data processing" is valid for GPUs as well! On GPUs, this can be quantified with the arithmetic intensity (number of flops per texture lookup) of implementations.

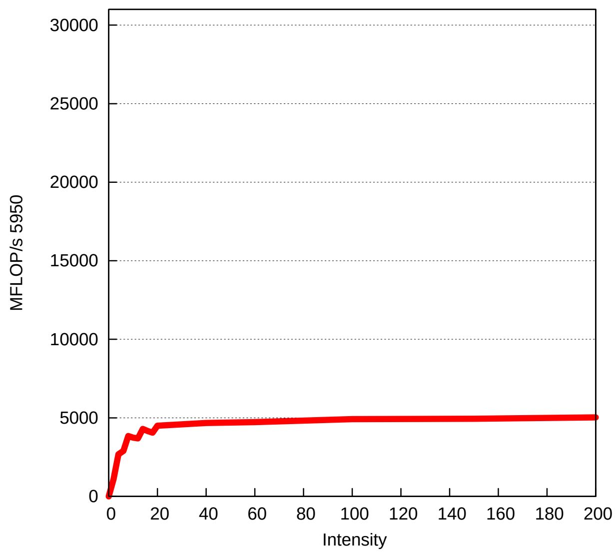
Test case:

- Fetch value x_i from long vector / texture $\mathbf{x}$, $i = 1, \ldots, 1024^2$.
- Compute $x_i = x_i + x_i^2 + x_i^3 + x_i^4 + \ldots + x_i^m$.
- Rewrite Horner-style: degree m yields $2m - 1$ flops.

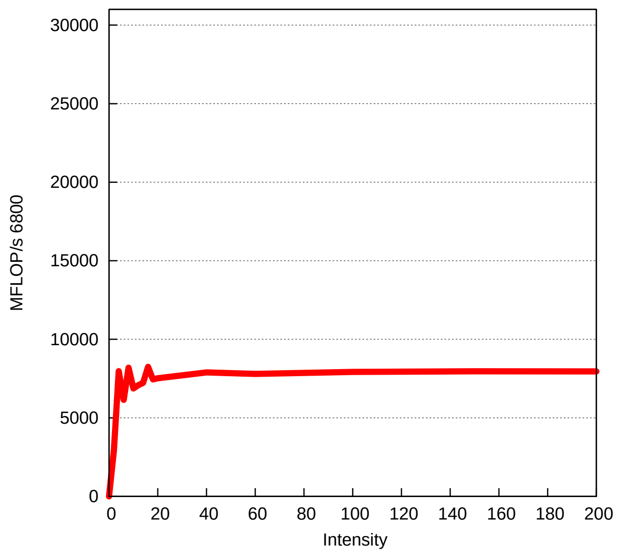
LS3

How to get closer to peak performance?

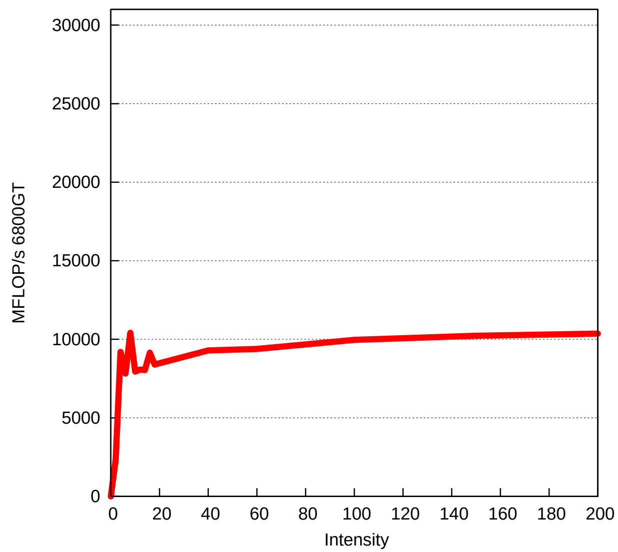
"Data moving is expensive, not data processing" is valid for GPUs as well! On GPUs, this can be quantified with the arithmetic intensity (number of flops per texture lookup) of implementations.



GF 5950



GF 6800



GF 6800GT*

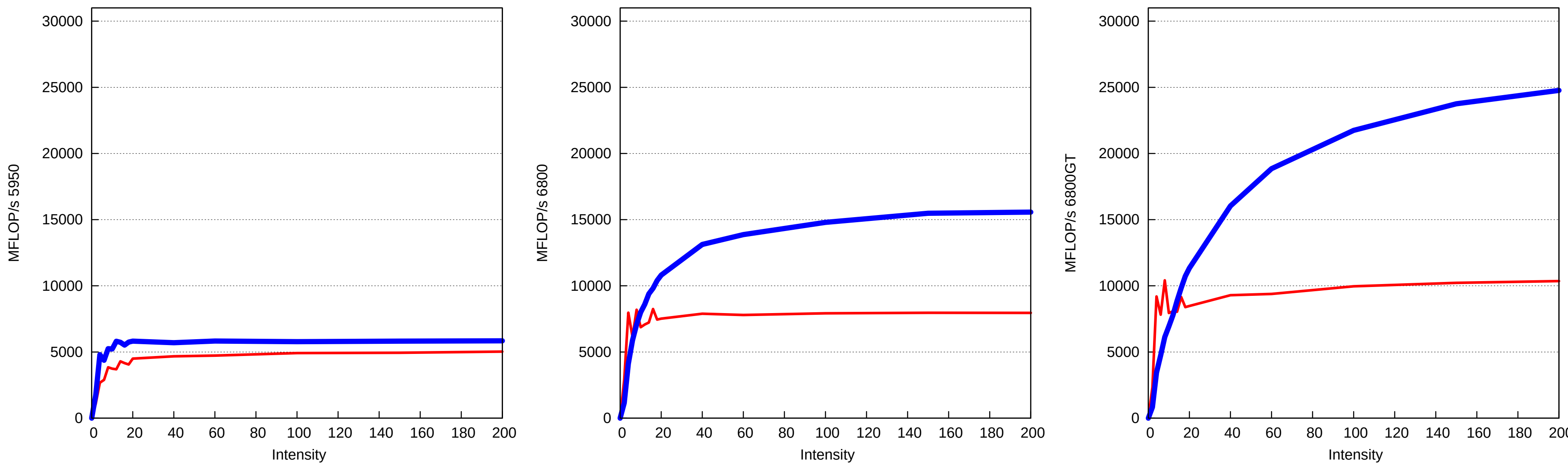
LUMINANCE

* courtesy of Hendrik Becker

LS3

How to get closer to peak performance?

”Data moving is expensive, not data processing” is valid for GPUs as well! On GPUs, this can be quantified with the arithmetic intensity (number of flops per texture lookup) of implementations.



GF 5950 GF 6800 GF 6800GT*

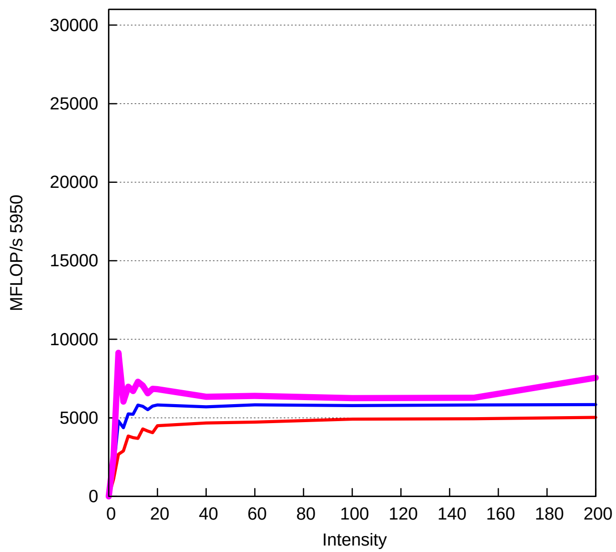
LUMINANCE RGBA32

* courtesy of Hendrik Becker

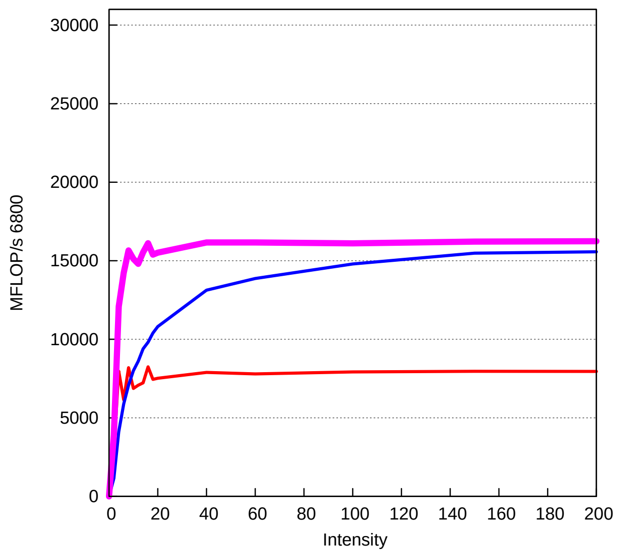
LS3

How to get closer to peak performance?

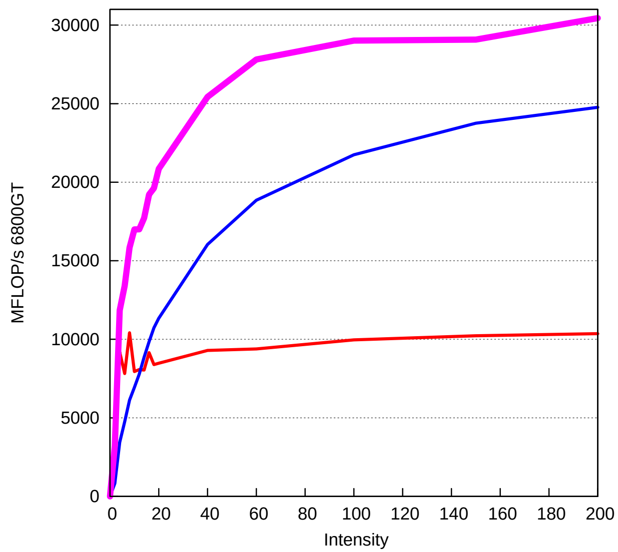
"Data moving is expensive, not data processing" is valid for GPUs as well! On GPUs, this can be quantified with the arithmetic intensity (number of flops per texture lookup) of implementations.



GF 5950



GF 6800



GF 6800GT*

LUMINANCE RGBA32 RGBA16

* courtesy of Hendrik Becker

LS3

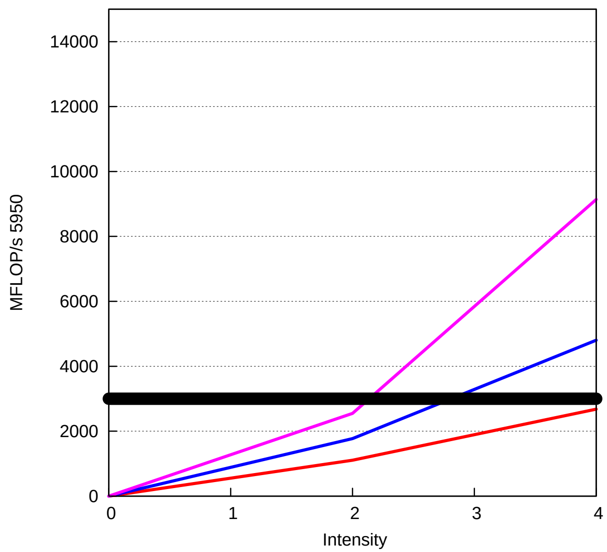
How to get closer to peak performance?

Realistic goal: 50% peak performance at a moderate intensity.

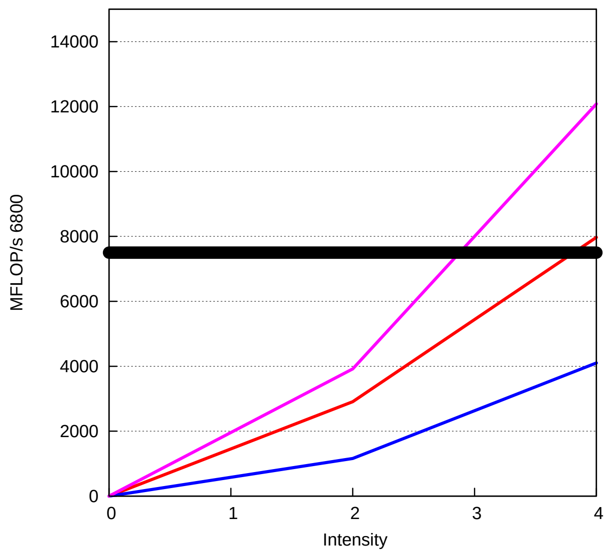
LS3

How to get closer to peak performance?

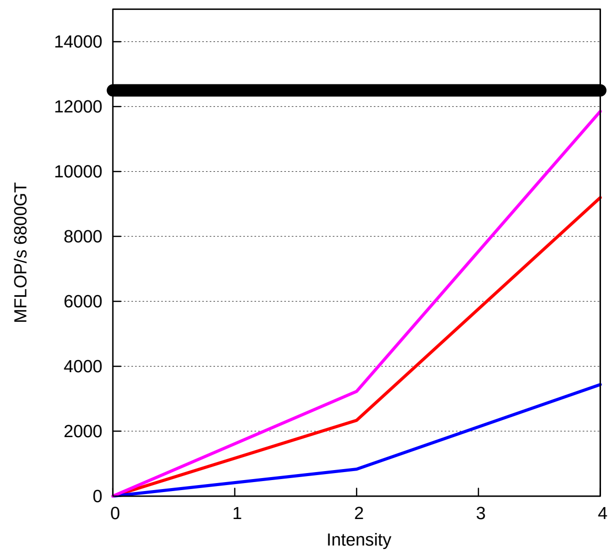
Realistic goal: 50% peak performance at a moderate intensity.



GF 5950



GF 6800



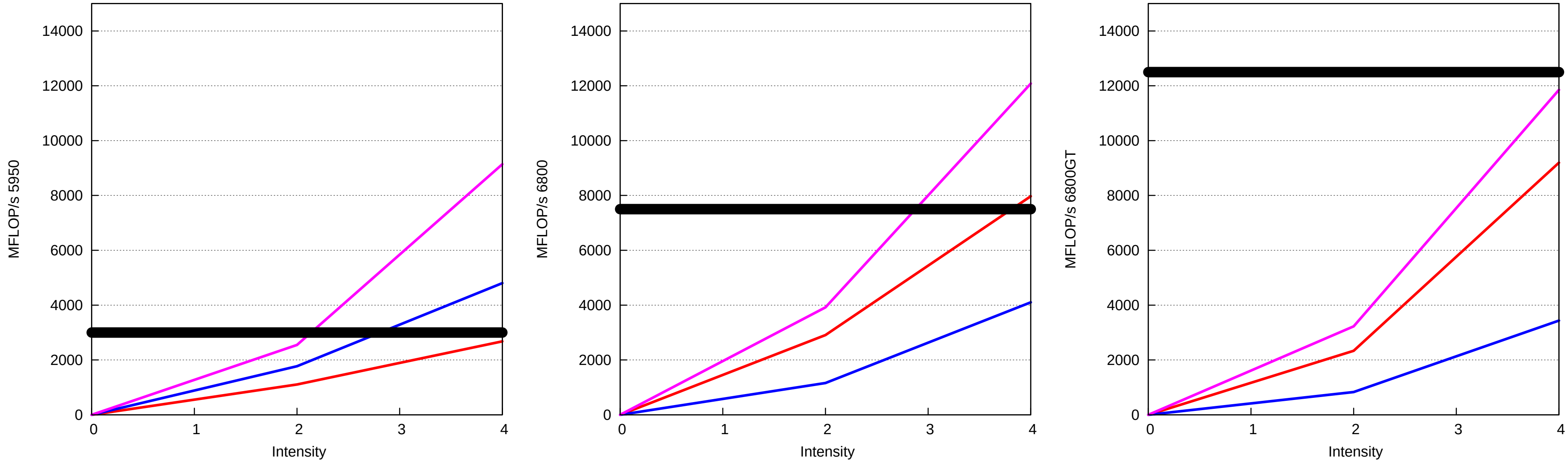
GF 6800GT

LUMINANCE RGBA32 RGBA16

LS3

How to get closer to peak performance?

Realistic goal: 50% peak performance at a moderate intensity.



GF 5950 GF 6800 GF 6800GT

LUMINANCE RGBA32 RGBA16

Intensity of all examples presented so far is ≈ 1! GFLOP/s rate for `MV_V` and `JACOBI` is within 90% of the measured peak for this intensity.

LS3

Conclusions: Towards numerical GIGAFLOP/s

Short term goal: Investigate further into using fast 16 bit preconditioning, this way working around the ”intensity barrier”.

Conclusions: Towards numerical GIGAFLOP/s

Short term goal: Investigate further into using fast 16 bit preconditioning, this way working around the ”intensity barrier”.

Long term goal: Try to reformulate core FEM-multigrid components to increase their intensity: Assemble on-the-fly? Smart, complex preconditioners like ILU, ADI and SPAI?

LS3

Conclusions: Towards numerical GIGAFLOP/s

Short term goal: Investigate further into using fast 16 bit preconditioning, this way working around the ”intensity barrier”.

Long term goal: Try to reformulate core FEM-multigrid components to increase their intensity: Assemble on-the-fly? Smart, complex preconditioners like ILU, ADI and SPAI?

Software goal: Don’t implement a full solver on the GPU, instead include the GPU as a fast preconditioner into the FEAST framework.

Conclusions: Towards numerical GIGAFLOP/s

Short term goal: Investigate further into using fast 16 bit preconditioning, this way working around the ”intensity barrier”.

Long term goal: Try to reformulate core FEM-multigrid components to increase their intensity: Assemble on-the-fly? Smart, complex preconditioners like ILU, ADI and SPAI?

Software goal: Don’t implement a full solver on the GPU, instead include the GPU as a fast preconditioner into the FEAST framework.

Questions? Comments? Your opinion?