
Massively Parallel Computing with Cuda

- Numerical Schemes -

Hendrik Lensch
Robert Strzodka

Today

- **Dense Linear Algebra / Libraries**
 - BLAS
 - FFT
- **Mixed Precision Methods**

Dense Linear Algebra / Libraries

CUBLAS

CUBLAS is an implementation of **BLAS (Basic Linear Algebra Subprograms)** on top of the CUDA driver.

The library is **self-contained at the API level**, that is, no direct interaction with the CUDA driver is necessary.

The basic model by which applications use the CUBLAS library is to:

- **create** matrix and vector objects in GPU memory space,
- **fill** them with data,
- **call** a sequence of CUBLAS functions,
- **download** the results from GPU memory space back to the host.

CUBLAS provides helper functions for **creating and destroying** objects in GPU space, and for **writing data to and retrieving data** from these objects.

Supported features

- **BLAS functions implemented (single precision only):**
 - **Real** data: Level 1, 2 and 3
 - **Complex** data: Level1 and CGEMM

- **Level 1=vector vector $O(N)$**
- **Level 2= matrix vector $O(N^2)$**
- **Level 3=matrix matrix $O(N^3)$**

- **For maximum compatibility with existing Fortran environments, CUBLAS uses **column-major storage**, and **1-based indexing**:**

Since C and C++ use row-major storage, this means applications cannot use the native C array semantics for two-dimensional arrays. Instead, macros or inline functions should be defined to implement matrices on top of one-dimensional arrays.

Using CUBLAS

- The interface to the CUBLAS library is the header file **cublas.h**
- Function names: **cublas(Original name)**
e.g. **cublasSGEMM**
- Because the CUBLAS core functions (as opposed to the helper functions) **do not return error status directly**, CUBLAS provides a separate function to retrieve the last error that was recorded, to aid in debugging
- cublasStatus cublasGetError()**

Returns the last error that occurred on invocation of any of the CUBLAS core functions. Reading the error status via `cublasGetError()` resets the internal error state to `CUBLAS_STATUS_SUCCESS`.

cublasInit, cublasShutdown

cublasStatus cublasInit()

Initializes the CUBLAS library and must be called before any other CUBLAS API function is invoked.

It allocates hardware resources necessary for accessing the GPU.

cublasStatus cublasShutdown()

Releases CPU-side resources used by the CUBLAS library.

The release of GPU-side resources may be deferred until the application shuts down.

cublasAlloc, cublasFree

cublasStatus cublasAlloc (int n, int elemSize, void **devicePtr)

Creates an object in GPU memory space capable of **holding n elements**, where each element requires **elemSize bytes** of storage.

Note that this is a **device pointer** that cannot be dereferenced in host code. cublasAlloc() is a wrapper around cudaMalloc().

Device pointers returned by cublasAlloc() can therefore be **passed to any CUDA device kernels**, not just CUBLAS functions.

cublasStatus cublasFree(const void *devicePtr)

Destroys the object in GPU memory space referenced by devicePtr.

cublasSetVector, cublasGetVector

**cublasStatus cublasSetVector(int n, int elemSize, const void *x,
int incx, void *y, int incy)**

Copies n elements from a vector x in CPU memory space to a vector y in GPU memory space.

Elements in both vectors are assumed to have a size of elemSize bytes.

Storage spacing between consecutive elements is incx for the source vector x and incy for the destination vector y

**cublasStatus cublasGetVector(int n, int elemSize, const void *x,
int incx, void *y, int incy)**

Copies n elements from a vector x in GPU memory space to a vector y in CPU memory space.

cublasSetMatrix, cublasGetMatrix

**cublasStatus cublasSetMatrix(int rows, int cols, int elemSize,
const void *A, int lda, void *B, int ldb)**

Copies a tile of *rows x cols* elements from a matrix A in CPU memory space to a matrix B in GPU memory space.

Each element requires storage of elemSize bytes.

Both matrices are assumed to be stored in **column-major format**, with the **leading dimension (that is, the number of rows)** of source matrix A provided in lda, and the leading dimension of destination matrix B provided in ldb.

**cublasStatus cublasGetMatrix(int rows, int cols, int elemSize,
const void *A, int lda, void *B, int ldb)**

Copies a tile of *rows x cols* elements from a matrix A in GPU memory space to a matrix B in CPU memory space.

cublasSgemm Example

```
#include <stdio.h>
#include <stdlib.h>
#include "cublas.h"

int main(void)
{
    float *a_h, *b_h, *c_h;
    float *a_d, *b_d, *c_d;
    float alpha = 1.0f, beta = 0.0f;
    int N = 2048, n2 = N*N;
    int nBytes, i;

    nBytes = n2*sizeof(float);

    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    c_h = (float *)malloc(nBytes);

    for (i=0; i < n2; i++) {
        a_h[i] = rand() / (float) RAND_MAX;
        b_h[i] = rand() / (float) RAND_MAX;
    }

    cublasInit();

    cublasAlloc(n2, sizeof(float), (void **)&a_d);
    cublasAlloc(n2, sizeof(float), (void **)&b_d);
    cublasAlloc(n2, sizeof(float), (void **)&c_d);

    cublasSetVector(n2, sizeof(float), a_h, 1, a_d, 1);
    cublasSetVector(n2, sizeof(float), b_h, 1, b_d, 1);

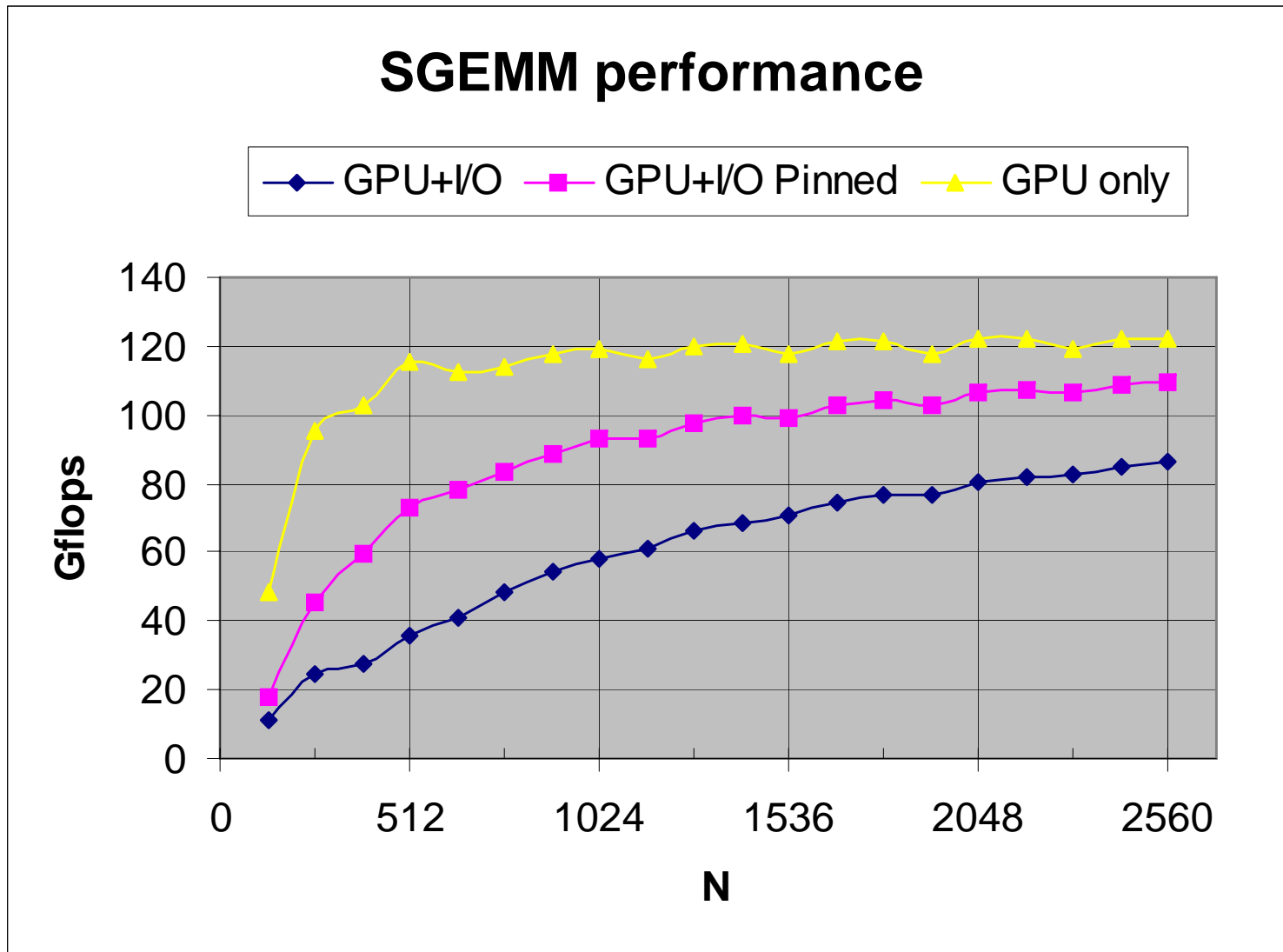
    cublasSgemm('n', 'n', N, N, N, alpha, a_d, N,
                b_d, N, beta, c_d, N);

    cublasGetVector(n2, sizeof(float), c_d, 1, c_h, 1);

    free(a_h); free(b_h); free(c_h);
    cublasFree(a_d); cublasFree(b_d);
    cublasFree(c_d);

    cublasShutdown();
    return 0;
}
```

CUBLAS performance



CUFFT

CUFFT is an implementation of the **Fast Fourier Transform (FFT)** .

It is a **divide-and-conquer algorithm** for efficiently computing discrete Fourier transform of complex or real-valued data sets.

The FFT is one of the most important and widely used numerical algorithms.

Supported features

- **1D, 2D and 3D transforms** of complex and real-valued data
- **Batched execution** for doing multiple 1D transforms in parallel
- **1D transform size up to 8M elements**
- **2D and 3D transform sizes in the range [2,16384]**
- **In-place and out-of-place** transforms for **real** and **complex** data.

CUFFT Types and Definitions

type cufftHandle:

is a handle type used to store and access CUFFT plans

type cufftResults:

is an enumeration of values used as API function values return values.

CUFFT_SUCCESS

Any CUFFT operation is successful.

CUFFT_INVALID_PLAN

CUFFT is passed an invalid plan handle.

CUFFT_ALLOC_FAILED

CUFFT failed to allocate GPU memory.

CUFFT_INVALID_TYPE

The user requests an unsupported type.

CUFFT_INVALID_VALUE

The user specifies a bad memory pointer.

CUFFT_INTERNAL_ERROR

Used for all internal driver errors.

CUFFT_EXEC_FAILED

CUFFT failed to execute an FFT on the GPU.

CUFFT_SETUP_FAILED

The CUFFT library failed to initialize.

CUFFT_SHUTDOWN_FAILED

The CUFFT library failed to shut down.

CUFFT_INVALID_SIZE

The user specifies an unsupported FFT size.

Transform types

- The library supports **complex and real** data transforms:
CUFFT_C2C, CUFFT_C2R, CUFFT_R2C
with directions:
CUFFT_FORWARD (-1) and CUFFT_BACKWARD (1)
- For complex FFTs, the input and output arrays **must interleave the real and imaginary part** (cufftComplex type is defined for this purpose)
- For real-to-complex FFTs, the output array holds only the non-redundant complex coefficients:
N \rightarrow N/2+1
N₀ x N₁ x ... x N_n \rightarrow N₀ x N₁ x ... x (N_n/2+1)
To perform in-place transform the input/output needs to be padded

More on transforms

- For 2D and 3D transforms, CUFFT performs transforms in **row-major order (C-order)**.
- CUFFT performs un-normalized transforms:
$$\text{IFFT}(\text{FFT}(A)) = \text{length}(A) * A$$
- CUFFT API is modeled after FFTW. Based on **plans, that completely specify the optimal configuration to execute a particular size of FFT**.
- Once a plan is created, the library stores whatever **state is needed to execute the plan multiple times without recomputing the configuration**: it works very well for CUFFT, because different kinds of FFTs require different thread configurations and GPU resources.

cufftPlan1d()

cufftResult cufftPlan1d(cufftHandle *plan, int nx, cufftType type, int batch);

Creates a 1D FFT plan configuration for a specified signal size and data type. The [batch input parameter](#) tells CUFFT how many 1D transforms to configure.

Input:

- plan Pointer to a cufftHandle object
- nx The transform size (e.g., 256 for a 256-point FFT)
- type The transform data type (e.g., CUFFT_C2C for complex-to-complex)
- batch Number of transforms of size nx

Output:

- plan Contains a CUFFT 1D plan handle value

cufftPlan2d()

cufftResult cufftPlan2d(cufftHandle *plan, int nx, int ny, cufftType type);

Creates a 2D FFT plan configuration for a specified signal size and data type.

Input:

- plan Pointer to a cufftHandle object
- nx The transform size in X dimension
- ny The transform size in Y dimension
- type The transform data type (e.g., CUFFT_C2C for complex-to-complex)

Output:

- plan Contains a CUFFT 2D plan handle value

cufftPlan3d()

```
cufftResult cufftPlan3d(cufftHandle *plan, int nx, int ny, int nz, cufftType type );
```

Creates a 3D FFT plan configuration for a specified signal size and data type.

Input:

- plan Pointer to a cufftHandle object
- nx The transform size in X dimension
- ny The transform size in Y dimension
- nz The transform size in Z dimension
- type The transform data type (e.g., CUFFT_C2C for complex-to-complex)

Output:

- plan Contains a CUFFT 3D plan handle value

cufftDestroy(),

cufftResult cufftDestroy(cufftHandle plan);

Frees all GPU resources associated with a CUFFT plan and destroys the internal plan data structure.

This function should be called once a plan is no longer needed to avoid wasting GPU memory.

Input:

plan cufftHandle object

cufftExecC2C()

```
cufftResult cufftExecC2C(cufftHandle plan,  
                        cufftComplex *idata, cufftComplex *odata,  
                        int direction);
```

Executes a CUFFT **complex to complex** transform plan.

CUFFT uses as **input data** the GPU memory pointed to by the **idata** parameter.

This function stores the **Fourier coefficients** in the **odata** array.

If **idata** and **odata** are the same, this method does an **in-place transform**.

Input:

plan	cufftHandle object for the plane to update
idata	Pointer to the input data (in GPU memory) to transform
odata	Pointer to the output data (in GPU memory)
direction	The transform direction (CUFFT_FORWARD or CUFFT_BACKWARD)

Output:

odata	Contains the complex Fourier coefficients
-------	---

cufftExecR2C()

```
cufftResult cufftExecR2C(cufftHandle plan,  
                        cufftReal *idata, cufftComplex *odata);
```

Executes a CUFFT **complex to complex** transform plan.

CUFFT uses as **input data** the GPU memory pointed to by the **idata** parameter.

This function stores the **Fourier coefficients** in the **odata** array.

If **idata** and **odata** are the same, this method does an **in-place transform**.

The output hold only the non-redundant complex Fourier coefficients.

Input:

plan	Pointer to a cufftHandle object
idata	Pointer to the input data (in GPU memory) to transform
odata	Pointer to the output data (in GPU memory)

Output:

odata	Contains the complex Fourier coefficients
-------	---

cufftExecC2R()

```
cufftResult cufftExecC2R(cufftHandle plan,  
                        cufftComplex *idata, cufftReal *odata);
```

Executes a CUFFT **complex to complex** transform plan.

CUFFT uses as **input data** the GPU memory pointed to by the **idata** parameter.

This function stores the **Fourier coefficients** in the **odata** array.

If **idata** and **odata** are the same, this method does an **in-place transform**.

The input hold only the non-redundant complex Fourier coefficients.

Input:

plan	Pointer to a cufftHandle object
idata	Pointer to the complex input data (in GPU memory) to transform
odata	Pointer to the real output data (in GPU memory)

Output:

odata	Contains the real-valued Fourier coefficients
-------	---

Example: 2D transform

```
#define NX 256
#define NY 128

cufftHandle plan;  cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);

/* Create a 2D FFT plan. */
cufftPlan2d(&plan, NX, NY, CUFFT_C2C);

/* Use the CUFFT plan to transform the signal out of place. */
cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

/* Inverse transform the signal in place. */
cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

/* Note: Different pointers to input and output arrays implies out of place transformation */

/* Destroy the CUFFT plan. */
cufftDestroy(plan);

cudaFree(idata), cudaFree(odata);
```

Performance

The CUFFT library implements **several FFT algorithms**, each with different performances and accuracy.

The **best performance** paths correspond to transform sizes that:

1. Fit in CUDA's shared memory
2. Are powers of a single factor (e.g. power-of-two)

If only condition 1 is satisfied, CUFFT uses a **more general mixed-radix** factor algorithm that is slower and less accurate numerically.

If none of the above conditions is satisfied, CUFFT uses **an out-of-place, mixed-radix algorithm** that stores all intermediate results in global GPU memory.

One notable exception is for long 1D transforms, where CUFFT uses a distributed algorithm that perform 1D FFT using 2D FFT.

CUFFT does not implement any specialized algorithms for real data, and so there is no direct performance benefit to using real

References

- **CUDA libraries**
 - See doc folder of CUDA toolkit
- **More on BLAS libraries**
 - Tomov et al. “Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems”
 - <http://www.netlib.org/lapack/lawnspdf/lawn210.pdf>
- **More on FFT**
 - Nukada et al. “Bandwidth Intensive 3-D FFT kernel for GPUs using CUDA”, Supercomputing 2008
 - Govindaraju et al. “High Performance Discrete Fourier Transforms on Graphics Processors”, Supercomputing 2008

Mixed Precision Methods

What is a Mixed Precision Method?

- **Definition:** A method that uses **different precisions** in its computations
- **Example:** **double**(a) + **double**(**float**(b) + **float**(c))
- **Typical usage:** Mix of **single** and **double** precision floating point computations
- **Goal:** Obtain the **same accuracy** but **better performance** with more low precision computations

Mixed Precision Performance Gains

- **Bandwidth bound algorithm**

- 64 bit = 1 double = 2 floats
- More variables per **bandwidth** (comp. intensity up)
- More variables per **storage** (data block size up)
- Applies to all **memory levels**:
disc → main → device → local → register

- **Computation bound algorithm**

- 1 double **multiplier** \approx 4 float **multiplier** (quadratic)
- 1 double **adder** \approx 2 float **adder** (linear)
- Multipliers are much bigger than adders
→ **Quadrupled** computational efficiency

Overview

- Why Bother with Mixed Precision?
- **Precision and Accuracy**
- Floating Point Operations
- Mixed Precision Iterative Refinement

Roundoff and Cancellation

Roundoff examples for the **float s23e8** format

additive roundoff $a = 1 + 0.00000004 = 1.00000004 =_{fl} 1$
multiplicative roundoff $b = 1.0002 * 0.9998 = 0.99999996 =_{fl} 1$
cancellation $c \in \{a, b\}$ $(c - 1) * 10^8 = \pm 4 =_{fl} 0$

Cancellation promotes the small error 0.00000004 to the absolute error 4 and a relative error of order one.

Order of operations can be crucial:

$$1 + 0.00000004 - 1 =_{fl} 0$$
$$1 - 1 + 0.00000004 =_{fl} 0.00000004$$

With the **double s52e11** format no problems above, but ...

An Instructive Example

Evaluating $f(x,y)$ with powers as multiplications [S.M. Rump, 1988]

$$f(x, y) = (333.75 - x^2) y^6 + x^2 (11x^2 y^2 - 121y^4 - 2) + 5.5y^8 + x/(2y)$$

for $x_0 = 77617$, $y_0 = 33096$ gives

float s23e8	1.1726
double s52e11	1.17260394005318
long double s63e15	1.172603940053178631

This is all wrong, even the sign is wrong!!

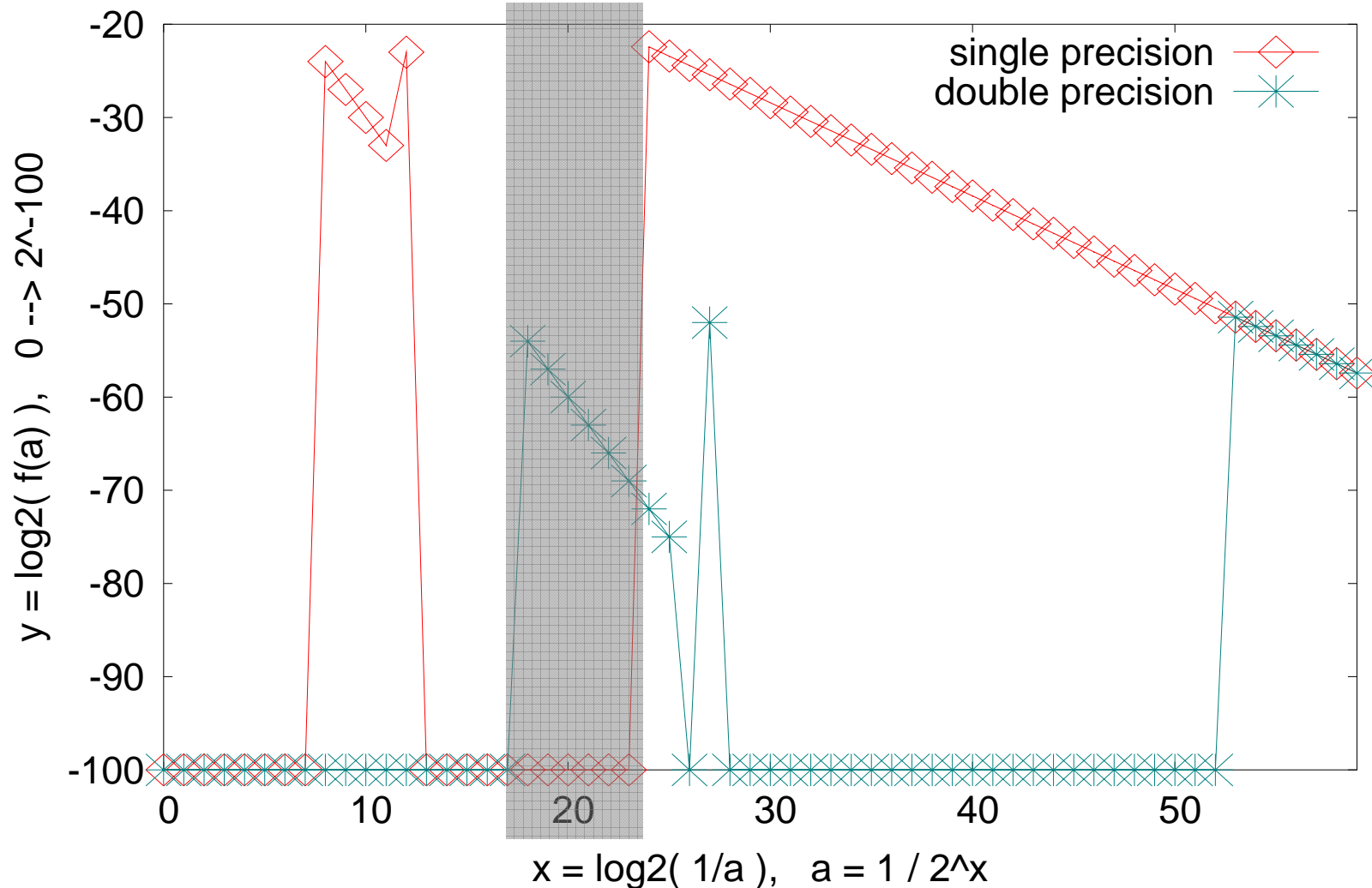
-0.82739605994682136814116509547981629...

Lesson learnt: **Computational Precision** \neq **Accuracy of Result**

The Erratic Roundoff Error

Roundoff error for: $0 = f(a) := |(1+a)^3 - (1+3a^2) - (3a+a^3)|$

← Smaller is better ←



The Dominant Data Error

- **Data error occurs when the **exact value** has to be **truncated** for storage in the binary format, e.g.**
 - π , $\sqrt{2}$, $\sin(2)$, $\exp(2)$, $1/3$, ...
 - In fact, any value, e.g. 0.1, except combinations of 2^b
- **So **more precision** is usually **better** because**
 - for float s23e8: $1 + 4e-8 =_{fl} 1$
 - for double s52e11: $1 + 4e-15 =_{fl} 1$
- **How can float be better than double then?**
 - There is **no data error** in the operands
 - Alternatively, the errors **cancel out** themselves favorably

Overview

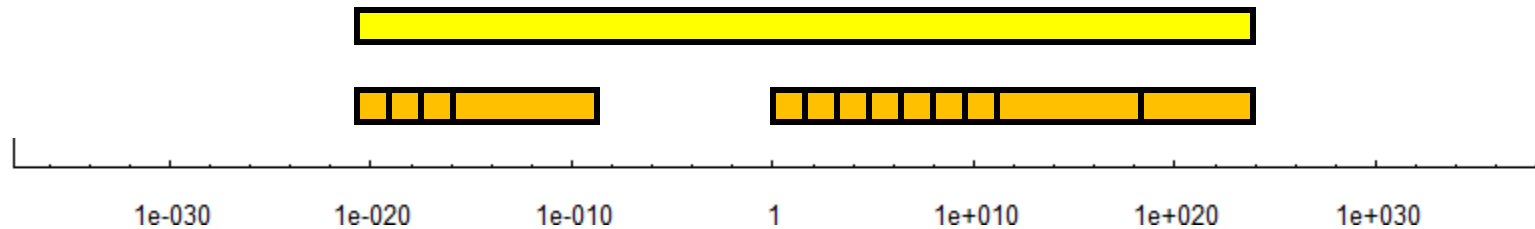
- Why Bother with Mixed Precision?
- Precision and Accuracy
- **Floating Point Operations**
- Mixed Precision Iterative Refinement

Understanding Floating Point Operations

- **Number representation s23e8**
 - $a = | 1\text{bit sign } s_a | 23\text{ bit mantissa } m_a | 8\text{ bit exponent } e_a |$
- **Multiplication $a * b$**
 - Operations: $s_a * s_b, m_a * m_b, e_a + e_b$
 - Exact format: $s46e9 = s23e8 * s23e8$
 - Main error: Mantissa truncated from 46 bit to 23 bit
- **Addition $a + b$**
 - Operations: $e_{\text{diff}} = e_a - e_b, m_a + (m_b \gg e_{\text{diff}}), \text{normalize}$
 - Exact format: $s278e8 = s23e8 + s23e8$
 - Main error: Mantissa truncated from 278 bit to 23 bit

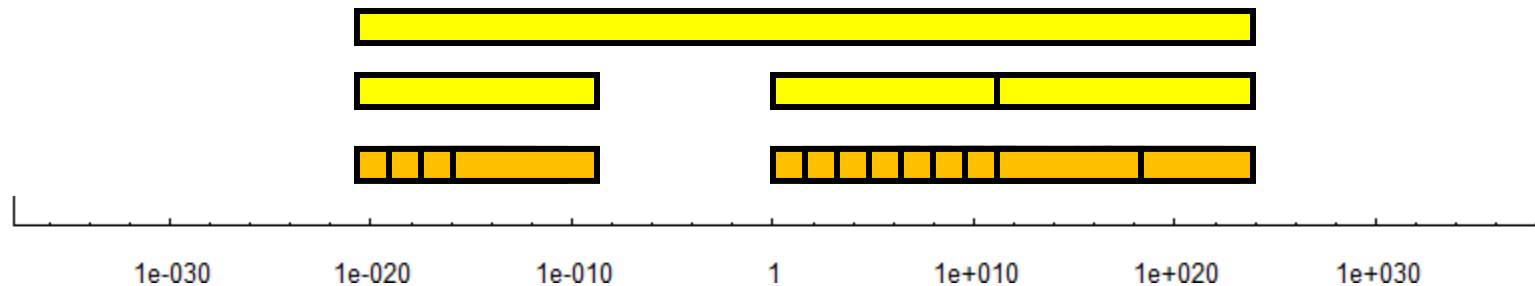
Commutative Summation

$$s = \sum_{i \in I} a_i$$



$$s = s_0 + s_1 + s_2$$

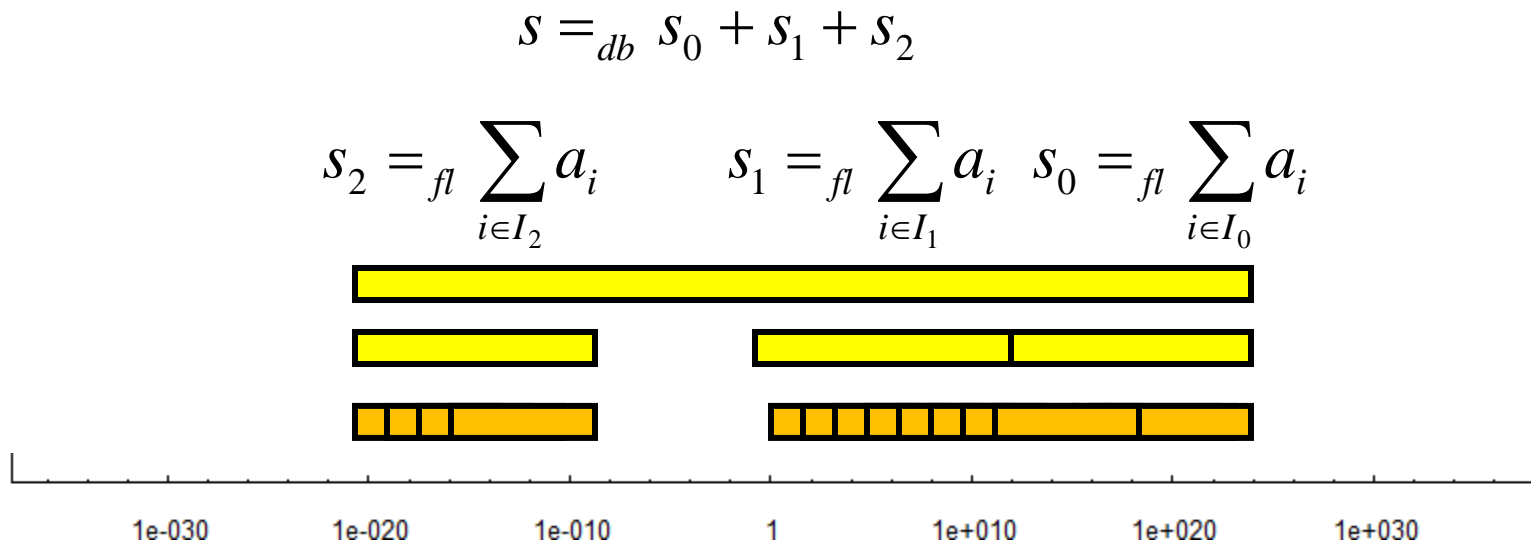
$$s_2 = \sum_{i \in I_2} a_i \quad s_1 = \sum_{i \in I_1} a_i \quad s_0 = \sum_{i \in I_0} a_i$$



Commutative Summation Example

- $1 + 0.000000004 =_{\text{db}} 1.000000004 =_{\text{fl}} 1$
- In **float s23e8**
 $s = \sum a_i = \frac{1}{2} + \frac{1}{2} + 0.000000004 - 0.000000003 =_{\text{fl}} 1$
- In **double s52e11**
 $s = \sum a_i =_{\text{db}} 1.000000001$
- In **mixed double/float**
 $s_0 = \sum_0 a_i = \frac{1}{2} + \frac{1}{2} =_{\text{fl}} 1$
 $s_1 = \sum_1 a_i = 0.000000004 - 0.000000003 =_{\text{fl}} 0.000000001$
 $s = s_0 + s_1 =_{\text{db}} 1.000000001$

Dependent Summation



$$s =_{db} \sum_{i \in I} a_i, \quad a_i = f_{db}(a_{i-1}), \quad \text{with slow double precision } f_{db}()$$

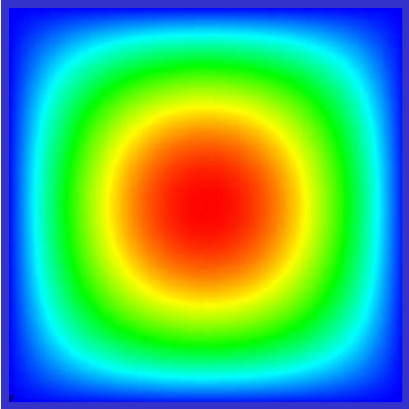
$$s_0 =_{fl} \sum_{i \in I_0} a_i, \quad s_1 =_{fl} \sum_{i \in I_1} a_i, \quad s_2 =_{fl} \sum_{i \in I_2} a_i, \quad s =_{db} s_0 + s_1 + s_2$$

$$a_i = f_{fl}(a_{i-1}), \quad \text{with fast single precision } f_{fl}()$$

Overview

- Why Bother with Mixed Precision?
- Precision and Accuracy
- Floating Point Operations
- **Mixed Precision Iterative Refinement**

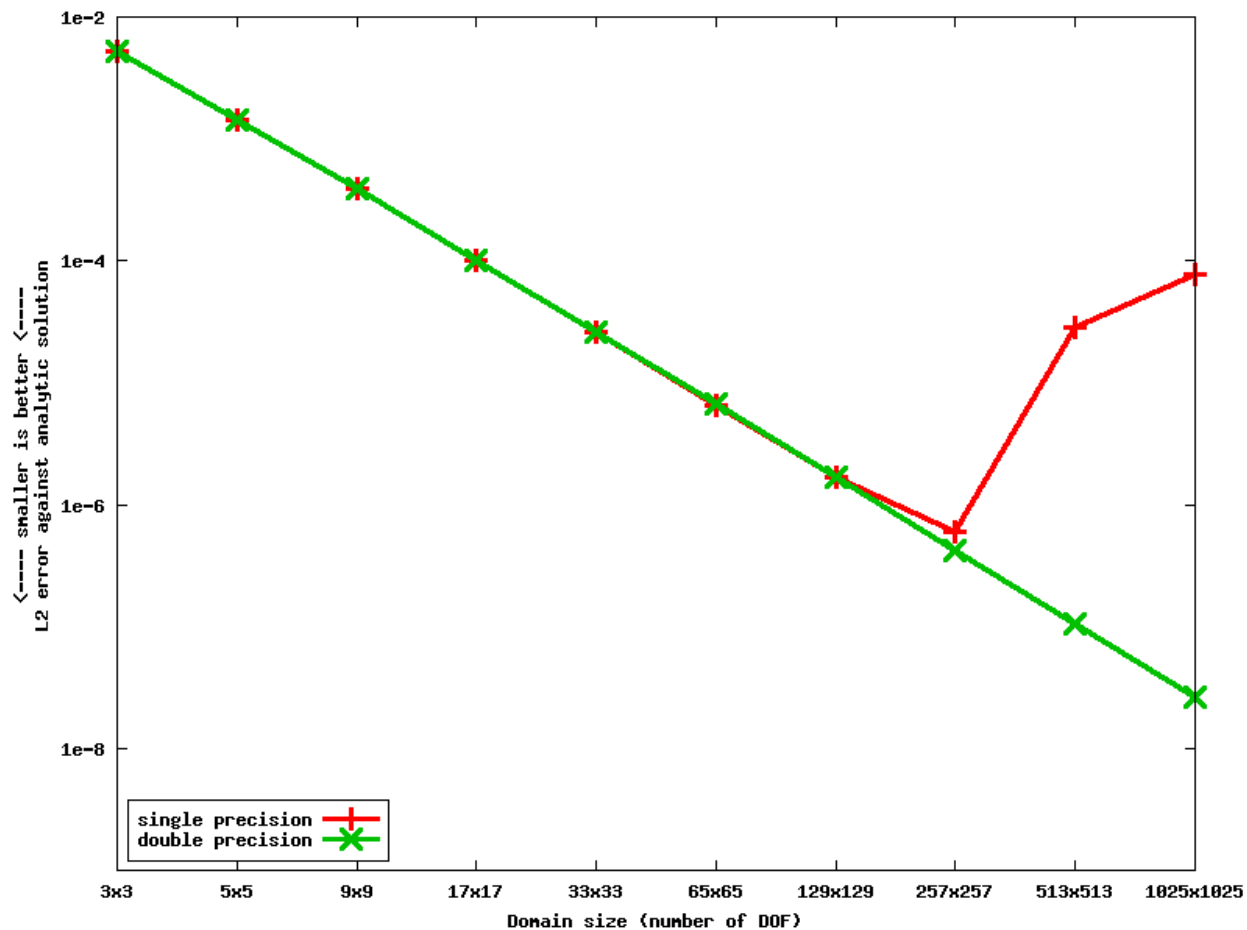
PDE Example: Poisson Problem

- $-\Delta u = f$
 - Unit square $[0,1]^2$
 - Bilinear conforming FEs (Q1)
 - Regular quadrilateral grid
 - Zero Dirichlet BCs
 - Analytic test function $x(1-x)y(1-y)$
- 
- Solved with multigrid until norms of residuals *indicate* convergence

PDE Example: Poisson Problem

- FEM theory: pure discretization error
- Expected error reduction of 4 (i.e. h^2) in each level

← Smaller is better ←



Mixed Precision Iterative Refinement

- **Exploit the speed of low precision and obtain a result of high accuracy**

$d_k = b - Ax_k$	Compute in high precision (cheap)
$Ac_k = d_k$	Solve in low precision (fast)
$x_{k+1} = x_k + c_k$	Correct in high precision (cheap)
$k = k+1$	Iterate until convergence in high precision

- **Low precision solution is used as a preconditioner in a high precision iterative method**
 - A is small and **dense**: Solve $Ac_k = d_k$ **directly**
 - A is large and **sparse**: Solve (approximately) $Ac_k = d_k$ with an **iterative** method itself

Direct Scheme for Small, Dense A

- **Algorithm**

- Compute $PA=LU$ once in single precision
- Use LU decomposition to solve $Ly=Pd_k, Uc_k=y$ in each step

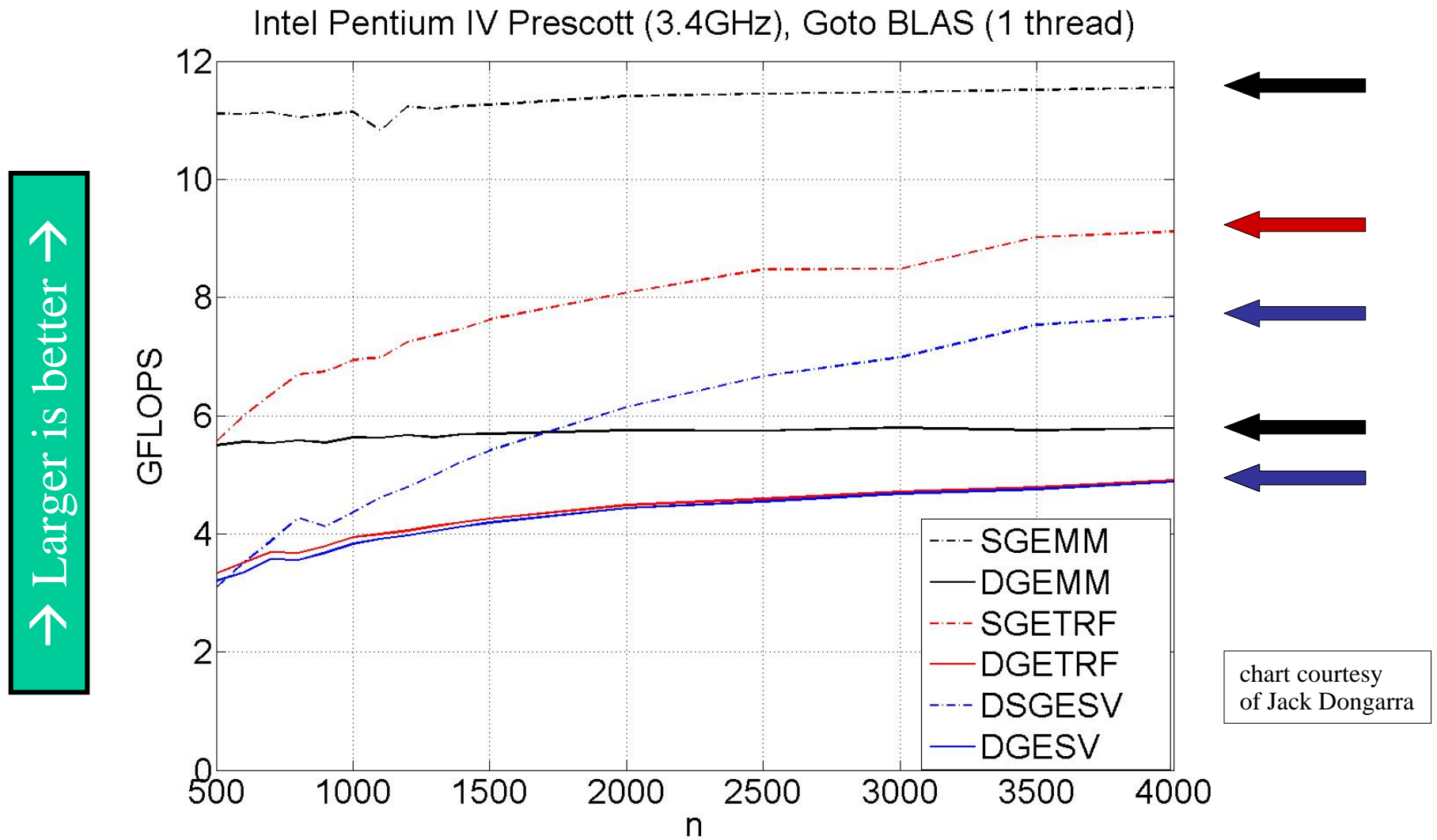
- **Main reasons for speedup**

- Computation of LU decomposition is $O(n^3)$
- Computation of LU is much faster in single than in double
- Solution using LU for several RHS is only $O(n^2)$

- **Upper bound for iteration count**

- $\text{ceil}(t_d/(t_s-K))$, where K, t_d, t_s are \log_{10} of matrix condition and double and single precision (e.g. t_d approx 16)

CPU SSE Results: LU Solver



Iterative Scheme for Large, Sparse A

- **Algorithm**

- Inner solver: **Conjugate Gradients, Multigrid**
- Correction loop can run on **CPU** or on **GPU** (old GPUs: emulated precision; new GPUs: true double precision)
- Terminate inner solver after **fixed number** of iterations, **fixed error reduction** or convergence

- **Main reason for speedup**

- Inner solver on the GPU runs almost at **peak bandwidth**

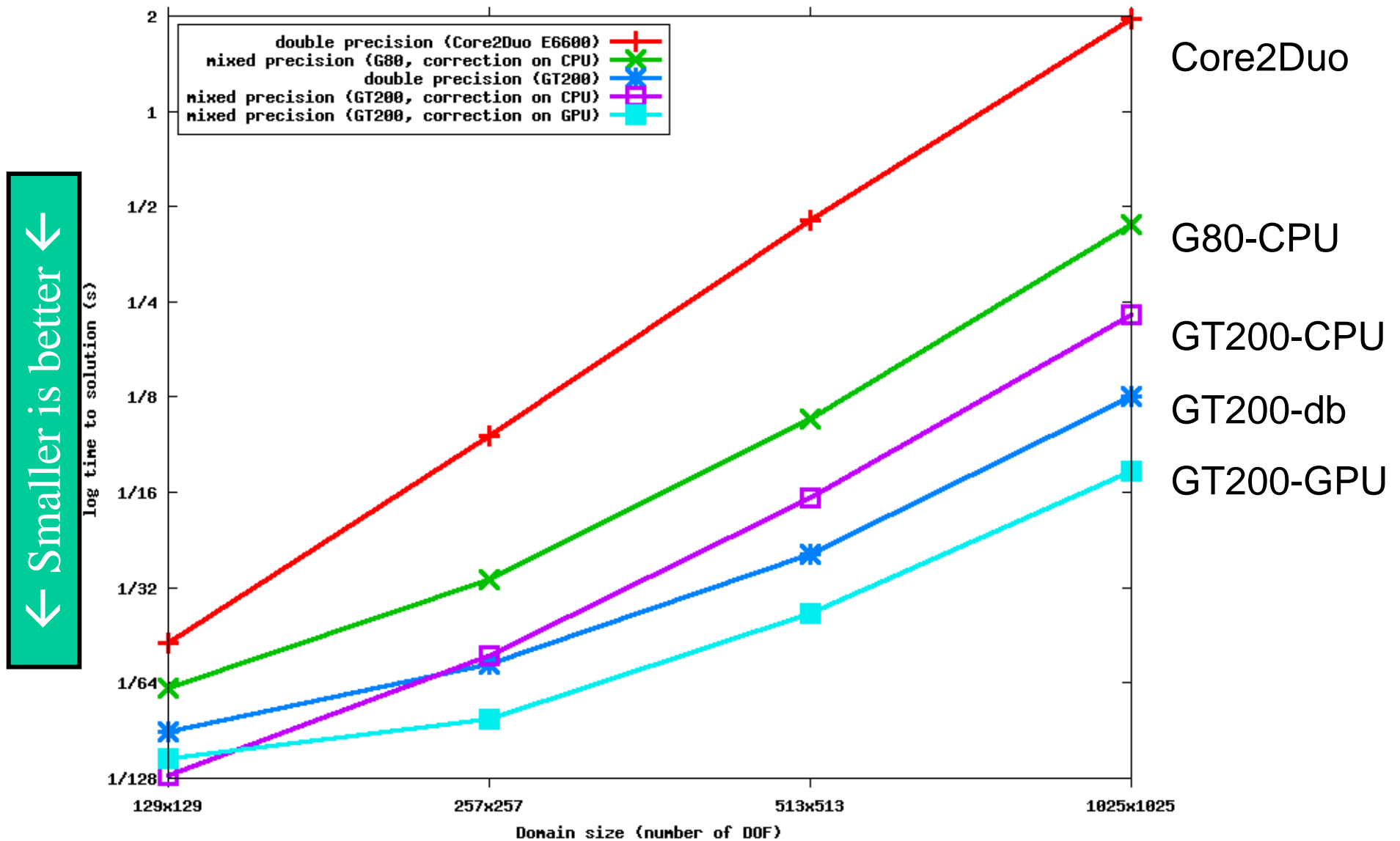
- **Applicability**

- Works even for very **ill-conditioned** matrices

GPU Performance Results

- **Test problem**
 - Poisson on unit square
 - Multigrid solver
 - $N=33^2$ to $N=1025^2$ DOF (=mesh points for Q1 FE)
- **Solver combination parameter space**
 - CPU implementation (Core2Duo E6600, SSE-optimized, double)
 - CUDA implementation (GeForce 8800 GTX and GeForce GTX 280)
 - Mixed precision, correction on CPU (G80 and GT200)
 - Native double precision (GT200 only)
 - Mixed precision, correction on GPU (GT200 only)

GPU Performance Results: CUDA



Summary

- The relation between **computational precision** and **final accuracy** is complicated but analyzable
- When single precision alone fails **iterative refinement** recovers the full accuracy with few double precision ops
- Mixed precision methods benefit **bandwidth** and even more **computation** bound algorithms
- Double precision GPUs are best utilized in mixed precision mode achieving **outstanding performance and accuracy**