

Linear algebra on multi-core and heterogeneous architectures

GPGPU

Web page:

<http://www.math.ethz.ch/~kressner/gpucomp.php>

Daniel Kressner

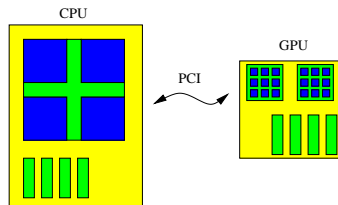
General purpose GPU computing

Contents:

- ▶ Programming GPUs – CUDA
- ▶ Matrix-Matrix Multiplies.
- ▶ Case study: QR factorization
- ▶ Case study: symmetric eigenvalue problems

GPU architecture, the rough guide

Very rough picture (almost sufficient for the purpose of this lecture):
Think of GPU as co-processor for outsourcing *heavy* level 3 BLAS operations.



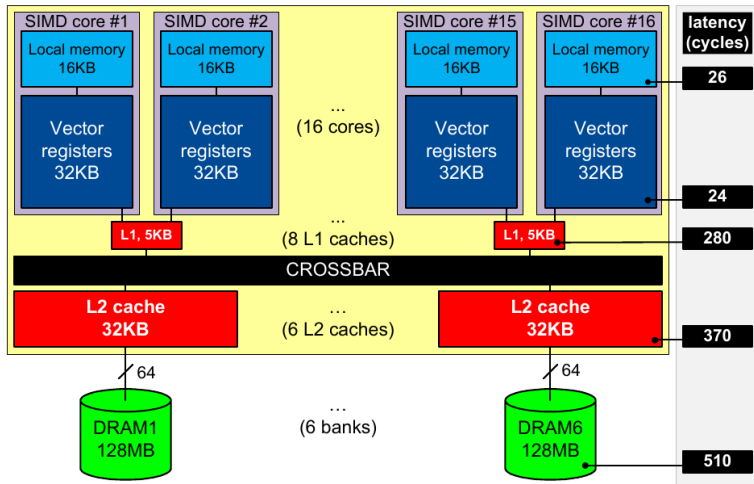
PCIe 1.1 interface for CPU↔GPU link. On our system, transferring 2000×2000 real SP matrix requires 4.5ms:

$$\text{bandwidth} \approx \frac{4\text{bytes} \times 2000^2}{4.5\text{ms}} \approx 3.5\text{GByte/s}$$

Memory latency about $11\mu\text{s}$ \rightsquigarrow transfer large chunks of memory
CPU↔GPU.

GPU architecture, a brief closer look

GeForce 8800GTX



GPU architecture, a brief closer look

GeForce 8800GTX:

- ▶ 128(!) scalar processors partitioned into 16 multiprocessors =: GPU core (each contains 8 procs)
- ▶ GPU cores have SIMD architecture.
- ▶ 32 scalar threads (e.g., 32 multiply-and-adds) are grouped into one so called **warp** (SIMD).
- ▶ requires 4 cycles to execute an entire warp.
- ▶ Runs at 1.35 GHz:

$$\text{peak performance} = 16 \times 32 \times 2 \times \frac{1.3 \times 10^9}{4} = 346 \text{ Gflops.}$$

GPU is a vector processor!

Memory management:

- ▶ Large number of registers, extremely small caches. 510 cycles latency to access main memory.

CUDA and CUBLAS

- ▶ CUDA (Compute Unified Device Architecture) programming environment (<http://www.nvidia.com/cuda>) tremendously helps programming NVIDIA GPUs.
- ▶ Extension of C; provides access to native instruction set and memory on NVIDIA GPUs.
- ▶ Fortran wrapper included. Third party Python and Java wrappers.
- ▶ Nontrivial to install, in particular under Linux: Requires proprietary driver by NVIDIA.
- ▶ CUBLAS: full implementation of BLAS within CUDA framework + extra routines for transferring matrices. Call with `cublas<BLAS>(. . .)`.

Outsourcing a matrix-matrix multiplication

C:

```
...
cublasAlloc(N*N, sizeof(d_A[0]), (void**)&d_A);
cublasSetVector(N*N, sizeof(h_A[0]), h_A, 1, d_A, 1);
...
cublasSgemm('n', 'n', N, N, N, alpha, d_A, N, d_B, N,
           beta, d_C, N);
cublasGetVector(N*N, sizeof(h_C[0]), d_C, 1, h_C, 1);
...
```

Fortran:

```
call cublas_init
cublas_alloc(N*N, sizeof_real, devPtrA)
call cublas_set_matrix (N, N, sizeof_real, A, N, devPtrA, N)
...
call cublas_SGEMM( 'N', 'N', N, N, N, ONE, devPtrA, N,
$                devPtrB, N, ONE, devPtrC, N )
call cublas_get_matrix (N, N, sizeof_real, devPtrC, N, C, N)
call cublas_free (devPtrA)
...
```

Our experimental setup

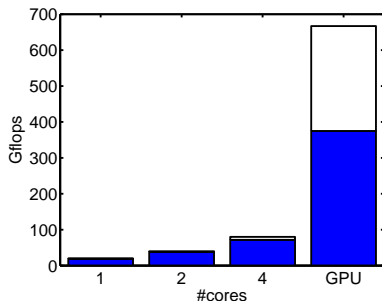
CPU = 2.5 GHz Intel Core2 Quad with 20 SP-Gflops/core; Intel MKL 10.1.

GPU = GeForce GTX 280 with 240 scalar processors: 667 SP-Gflops; CUDA 2.0.

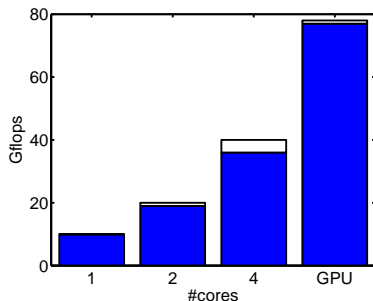
PCIe 1.1 interface for CPU↔GPU link (3.5 GByte/s effective bandwidth).

SGEMM GPU performance for $n = 2000$

SGEMM



DGEMM



It takes 220 milliseconds on 4 cores. It takes 40 milliseconds on GPU.
(not shown: plus 3×4.5 milliseconds for CPU-GPU transfer).

Profile CUDA by setting `CUDA_PROFILE=1`, creates log file `cuda_profile.log`:

```
method,gputime,cputime,occupancy
method=[ memcopy ] gputime=[ 4524.224 ]
method=[ memcopy ] gputime=[ 4524.000 ]
method=[ __globfunc__Z28fast_sgemm_gld_main_hw_na_nb17cublasSgemmParams ] ...
gputime=[ 40224.867 ] cputime=[ 16.000 ] occupancy=[ 1.000 ]
method=[ memcopy ] gputime=[ 4808.992 ]
```

BLAS 1 and small-sized BLAS 2 operations are bandwidth bound

- ▶ 2000×2000 matrix-vector multiplication (SGEMV) takes 0.761 milliseconds, these are 10 Gflops (not too bad).
- ▶ Transfer CPU→GPU takes 4.5 milliseconds (Gflops go down to 1.5 Gflops)!!

Does it make sense to compute scalar products at GPU at all?

Back-of-the-envelope calculation for 8800GTX

- ▶ Time for fetching vectors from main memory on GPU is

$$5\mu\text{s} + \frac{\text{Bandwidth required}}{75\text{GByte/s}}$$

- ▶ Scalar product performs $2n$ flops and requires to transfer $2n$ words = $8n$ bytes.
- ▶ $n = 20000 \rightsquigarrow 7\mu\text{s}$ for memory transfer but only $0.1\mu\text{s}$ for comp.
- ▶ For smaller n execution time nearly constant ($5\mu\text{s}$).
- ▶ For larger n , scalar product performance ultimately dominated by bandwidth ≤ 18 Gflops (unlikely attainable).

Case study: QR factorization

How to benefit **without too much pain** from GPU when performing a QR factorization? Two alternatives:

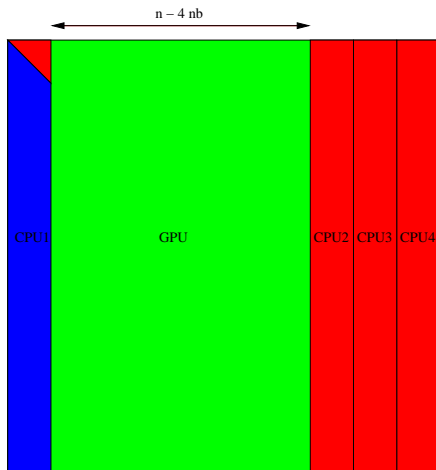
1. Use block QR (described in lecture on single cores) on GPU, by transferring entire matrix to GPU and replacing all calls in SGEQRF by calls to CUBLAS.
 - ▶ Updates rich in level 3 BLAS \rightsquigarrow **good**.
 - ▶ Panel factorization small-sized level 2, almost level 1 BLAS \rightsquigarrow **very bad**.
2. Use tiled block QR (described in lecture on multi-cores) on GPU. {Dis}advantages same as in 1.

Need new idea.

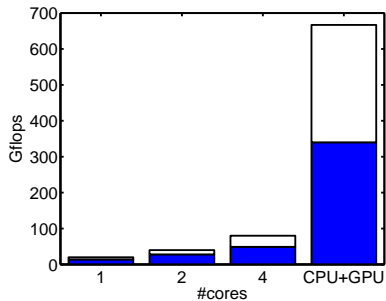
Case study: QR factorization

New idea:

- ▶ Use block QR but do **panel factorizations on CPU**.
- ▶ Optional: Do small part of update on CPU to keep all cores busy.



Measured GPU performance for $n = 4000$



Takes 1.7 seconds on 4 cores. Takes 0.25 seconds on CPU+GPU.
(including CPU-GPU transfer)

Case study: Symmetric eigenvalue problems

Compute all eigenvalues of square **symmetric matrix** A :

$$A = Q \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} Q^T, \quad Q \text{ orthogonal.}$$

Virtually all known dense methods proceed in two stages

1. Reduce A to tridiagonal form by orth similarity transformation.
2. Compute eigenvalues (optionally eigenvectors) of tridiagonal matrix (bisection, QR, divide-and-conquer, MRRR).

First stage computationally **far more expensive** than second (obvious if only eigenvalues are needed).

Reduction to tridiagonal form

Basic outline of algorithm:

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{Q_1} \begin{bmatrix} \times & \times & 0 & 0 \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{Q_2} \begin{bmatrix} \times & \times & 0 & 0 \\ \times & \times & \times & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

A $Q_1 A Q_1$ $Q_2 Q_1 A Q_1 Q_2$

$\rightsquigarrow T := Q^T A Q$ with $Q := Q_1 Q_2$. MATLAB script for computing T

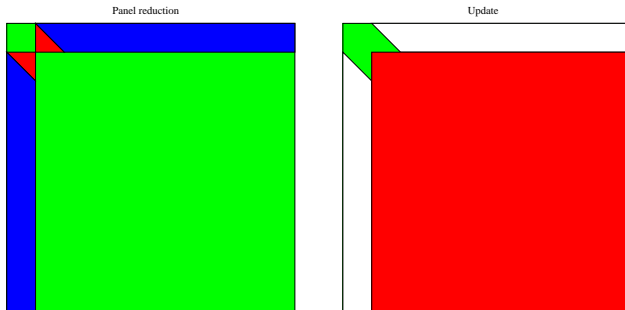
```
for k = 1:n-1,
    x = A(k+1:n,k); v = x;
    if x(1)>=0,
        v(1) = v(1) + norm(x);
    else
        v(1) = v(1) - norm(x);
    end
    v = v / v(1); tau = 2 / norm(v)^2;
    w = tau * A(k+1:n,k+1:n) * v; % level 2 BLAS (SSYMV)
    w = w - 1/2 * tau * (w'*v) * v; % level 1 BLAS (SAXPY)
    A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - v*w' - w*v';
                                     % level 2 BLAS (SSYR2)
end
```

Attains 3 Gflops (out of 20) for reducing 2000×2000 real SP matrix on single core.

Bringing level 3 BLAS to tridiagonal reduction

- ▶ LAPACK improves on scalar implementation, putting 50% in level 3 BLAS but 50% still performed in level 2 BLAS.
- ▶ Performs OK on single core (7.6 Gflops), but poorly on multi-cores (11.3 Gflops on 2 cores; 14.1 Gflops on 4 cores).

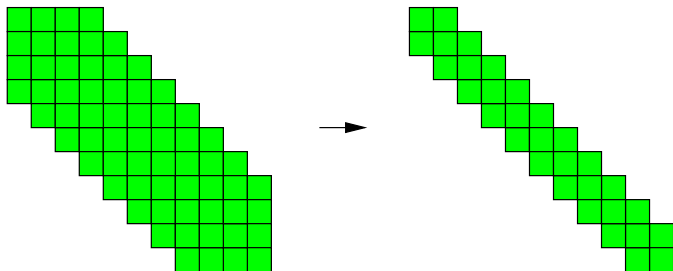
Alternative: Reduce first to block tridiagonal form (think of lower bandwidth $nb = 64$)



Panel reduction level 2 BLAS. Update level 3 BLAS (SYMM, SYR2K).

Reduction from block to tridiagonal form

Sweeping subdiagonal off (Schwarz algorithm):

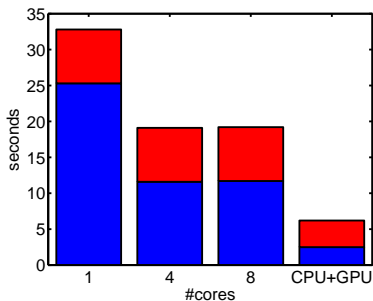


- ▶ $O(n^2)$ flops for $nb \ll n$ (but hard to make perform well on multi-core machines).
- ▶ Implemented in SBR (Successive Band Reduction) toolbox.

Measured execution times for $n = 6144$

CPU-GPU implementation:

- ▶ Level 3 BLAS performed on GPU.
- ▶ Some surprises: SYMM and SYR2K of CUBLAS 2.0 perform quite poorly (at most 15% of peak performance). Replaced by SGEMM, ignore symmetry in update.
- ▶ Reduction from block to tridiagonal nonnegligible.



Takes 19.1 seconds on 4 cores ($nb = 96$).

Takes 6.2 seconds on CPU+GPU ($nb = 32$).

Summary

- ▶ GPU easy to program with CUDA if existing kernels tuned can be used (CUBLAS, FFT, ...)
- ▶ Otherwise need to perform vectorization explicitly.
- ▶ Use of GPU improves performance of QR by factor 7, and of symmetric tridiagonal reduction by factor 3 on our experimental setup.
- ▶ Further reading:
Presentation draws heavily on various LAPACK working notes by Vasily Volkov+Jim Demmel and by Jack Dongarra et al. See <http://www.netlib.org/lapack/lawns/downloads/>.
Symmetric eigenvalue problems on GPUs ongoing joint work with Bientinesi, Igual, Quintana-Ortí, see preliminary preprint on website.