Efficient exact linear algebra over GPU

Michael Abshoff and Clément PERNET

SAGE Days 9,
August 15, 2008

Outline

Why we care

GPU's

Experimentations

Outline

Why we care

Exact computations:

- Number Theory: *modular forms* — $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}_p, GF(p^k)$
- Graph Theory: *graph isomorphism* — $GF(2)$
- Crypto: *NFS, Groebner basis* — $GF(2), GF(2^k)$
- ...

Why we care

Exact computations:

- Number Theory: *modular forms* $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}_p, GF(p^k)$
- Graph Theory: *graph isomorphism* $GF(2)$
- Crypto: *NFS, Groebner basis* $GF(2), GF(2^k)$
- ...

Boil down to Linear Algebra:

- Number Theory: echelon, charpoly
- Graph Theory: charpoly
- Crypto: solve, echelon form
- ...

Why we care

Exact computations:

- Number Theory: *modular forms* $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}_p, GF(p^k)$
- Graph Theory: *graph isomorphism* $GF(2)$
- Crypto: *NFS, Groebner basis* $GF(2), GF(2^k)$
- ...

Boil down to Linear Algebra:

- Number Theory: echelon, charpoly
- Graph Theory: charpoly
- Crypto: solve, echelon form
- ...

Mathematics is the art of reducing everything to Linear Algebra !

W. Stein

Efficient linear algebra

An example: Matrix multiplication

sage.math : Opteron 2.4Ghz

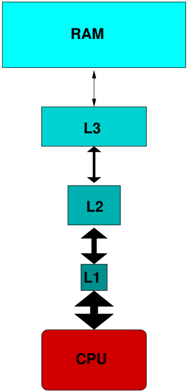
naive, triple loop	$n = 1000$	$\Rightarrow$23.8s
naive, triple loop, 1 line diff	$n = 1000$	$\Rightarrow$4.8s
BLAS	$n = 1000$	$\Rightarrow$0.7s

Optimizing the simplest operation in Linear Algebra is **not trivial**

Optimizing data locality

Memory considerations:

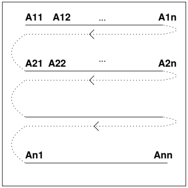
- CPU-Memory communication: bandwidth gap
  ⇒Hierarchy of several cache memory levels



Optimizing data locality

Memory considerations:

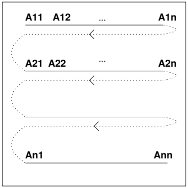
- CPU-Memory communication: bandwidth gap
  $\Rightarrow$Hierarchy of several cache memory levels
- Row major representation of matrices



Optimizing data locality

Memory considerations:

- CPU-Memory communication: bandwidth gap
  ⇒Hierarchy of several cache memory levels
- Row major representation of matrices
- a RAM memory access can fetch a bunch of contiguous elements



Optimizing data locality

Comparing

for i=1 to n **do**
  for j=1 to n **do**
    for k=1 to n **do**
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end for
  end for
end for

Optimizing data locality

Comparing

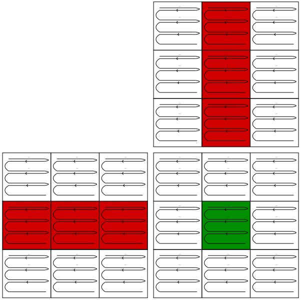
for i=1 to n **do**
  for j=1 to n **do**
    for k=1 to n **do**
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end for
  end for
end for

VS

for i=1 to n **do**
  for k=1 to n **do**
    for j=1 to n **do**
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end for
  end for
end for

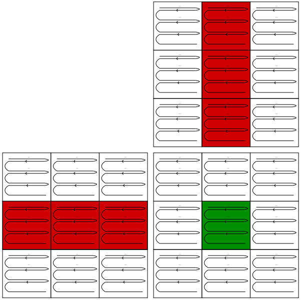
Further memory optimizations

Larger dimensions: cache blocking.
⇒split matrices into blocks, s.t. their product can be computed within the cache.



Further memory optimizations

Larger dimensions: cache blocking.
⇒split matrices into blocks, s.t. their product can be computed within the cache.



Reuse of the data

- if `Work` $\gg$ `Data` : memory fetch is amortized
  ⇒reach the peak performance of the CPU
- Matrix multiplication: $n^3 \gg n^2$
  ⇒well suited for block design

Arithmetic optimizations

- fma (fused multiply and accumulate) $z \leftarrow z + x * y$
- pipeline
- SSE
- ...

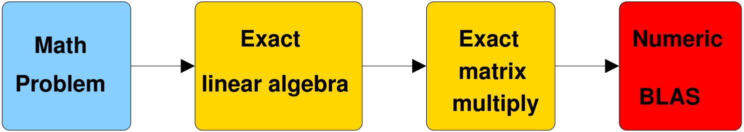
Arithmetic optimizations

- fma (fused multiply and accumulate) $z \leftarrow z + x * y$
- pipeline
- SSE
- ...

Tends to give advantage to floating point arithmetic up to now.

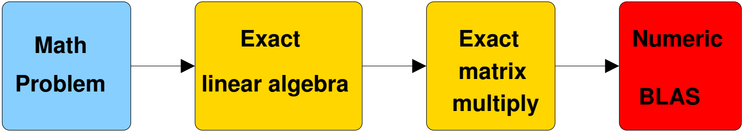
Overall approach

Reductions, reductions, reductions, ...:



Overall approach

Reductions, reductions, reductions, ...:



How to get faster ?

- parallel BLAS: ATLAS now scales linearly with the number or cores/CPUs
- Graphical Processing Units: GPU's

Outline

A history of the GPU technology

- “dumb” framebuffer
- Bitblt: copy interleaved rgb bitmaps quickly
- offloading of 3D computations to the GPU, i.e. z-buffers
- (primitive early) Shaders: small, up to 128 instructions, no branching, etc
- CuDa: C compiler that produces code running on the GPU

Interesting GPUs

- current NVidia: Tesla C870 GPU. 128 thread processors, 1.5 GB dedicated Memory
- Fall 2008 Nvidia: Tesla C1060. 240 thread processors, 4 GB dedicated memory at 102 GB/sec, 90 GFlops Double Precision, 360 GFlops Single Precision
- current ATI: RV770, 800 SPs, 1GB+ dedicated Memory, 1.2TFLOPS single precision, 150GFlops Double Precision
- Intel: Larrabee - A Many-Core x86 Architecture for Visual Computing (Vaporware, 1TFlop Single Precision)

Most of the above will/are conforming to IEEE specs. In comparison: Intel high end Core2 Quad: 100 GFlops Single Precision

GPU Alternatives

- FPGAs
- IBM's Cell CPU, especially the second generation
- CPU/GPU combos, i.e. AMD/ATi
- general Heterogeneous cluster hardware

GPU: Programming Tools

- CuDABLAS: Easy to hook into existing (numerical software)
- generic CuDA: Write C code, compile it to GPU code (this is NVidia specific)
- OpenCL
- Intel's Secret Sauce

GPU programming and Sage

- Sage will have CuDABLAS support as an optional package by Sage Days 10 in October in Nancy.
- generic CuDa support will also likely exist for certain well behaved computations like Monte Carlo Simultations.
- CuDa support is not for the faint of heart, i.e. driver issues cause a lot of problems.

Outline

Using CUDA BLAS interface to GPU

```
cublasAlloc(n*n,sizeof(*a),(void**)\&devPtrA);
cublasSetMatrix (n,n,sizeof(*a),a,n,devPtrA,n);

cublasAlloc(n*n,sizeof(*b),(void**)\&devPtrB);
cublasSetMatrix (n,n,sizeof(*b),a,n,devPtrB,n);

cublasAlloc(n*n,sizeof(*c),(void**)\&devPtrC);
cublasSetMatrix (n,n,sizeof(*c),c,n,devPtrC,n);

cublasSgemm ('N', 'N', n, n, n, 1.0,
             devPtrA, n,
             devPtrB, n,
             0.0, devPtrC, n);

cublasGetMatrix (n, n, sizeof(*c), devPtrC, n, c, n);

cublasFree(devPtrA);
cublasFree(devPtrB);
cublasFree(devPtrC);
```

Matrix multiplication over $\mathbb{Z}_{11}$
using BLAS sgemm (32 bits floats)

n	1000	1500	2000	2500
naive	8.0s	32.2s	82.1s	167s
naive + 1 line trick	1.9s	6.48s	15.4s	31.8s
GPU: CUDA	.65s	.97s	1.87s	4.28s
CPU: ATLAS (2cores)	.13s	.43s	1.07s	1.86s