Exact Linear Algebra Algorithmic: Theory and Practice
ISSAC’15 Tutorial

Clément Pernet

Université Grenoble Alpes, Inria, LIP-AriC

July 6, 2015
Matrices can be

**Dense:** store all coefficients

**Sparse:** store the non-zero coefficients only

**Black-box:** no access to the storage, only *apply* to a vector
Matrices can be

- **Dense**: store all coefficients
- **Sparse**: store the non-zero coefficients only
- **Black-box**: no access to the storage, only apply to a vector

Coefficient domains:

- **Word size**: integers with a priori bounds
  - $\mathbb{Z}/p\mathbb{Z}$ for $p$ of $\approx 32$ bits
- **Multi-precision**: $\mathbb{Z}/p\mathbb{Z}$ for $p$ of $\approx 100, 200, 1000, 2000, \ldots$ bits
- **Arbitrary precision**: $\mathbb{Z}, \mathbb{Q}$
- **Polynomials**: $K[X]$ for $K$ any of the above
Matrices can be

Dense: store all coefficients
Sparse: store the non-zero coefficients only
Black-box: no access to the storage, only apply to a vector

Coefficient domains:

Word size:
- integers with a priori bounds
- \( \mathbb{Z}/p\mathbb{Z} \) for \( p \) of \( \approx 32 \) bits

Multi-precision: \( \mathbb{Z}/p\mathbb{Z} \) for \( p \) of \( \approx 100, 200, 1000, 2000, \ldots \) bits

Arbitrary precision: \( \mathbb{Z}, \mathbb{Q} \)

Polynomials: \( K[X] \) for \( K \) any of the above

Several implementations for the same domain: better fits FFT, LinAlg, etc
Matrices can be

- **Dense**: store all coefficients
- **Sparse**: store the non-zero coefficients only
- **Black-box**: no access to the storage, only apply to a vector

**Coefficient domains:**

- **Word size**: integers with a priori bounds
  - \( \mathbb{Z}/p\mathbb{Z} \) for \( p \) of \( \approx 32 \) bits

- **Multi-precision**: \( \mathbb{Z}/p\mathbb{Z} \) for \( p \) of \( \approx 100, 200, 1000, 2000, \ldots \) bits

- **Arbitrary precision**: \( \mathbb{Z}, \mathbb{Q} \)

- **Polynomials**: \( K[X] \) for \( K \) any of the above

Several implementations for the same domain: better fits FFT, LinAlg, etc

Need to structure the design.
Motivations

Comp. Number Theory: CharPoly, LinSys, Echelon, over $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}$, Dense

Graph Theory: MatMul, CharPoly, Det, over $\mathbb{Z}$, Sparse

Discrete log.: LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 120$ bits, Sparse

Integer Factorization: NullSpace, over $\mathbb{Z}/2\mathbb{Z}$, Sparse

Algebraic Attacks: Echelon, LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 20$ bits, Sparse & Dense

List decoding of RS codes: Lattice reduction, over GF$(q)[X]$, Structured
Motivations

Comp. Number Theory: CharPoly, LinSys, Echelon, over $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}$, Dense

Graph Theory: MatMul, CharPoly, Det, over $\mathbb{Z}$, Sparse

Discrete log.: LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 120$ bits, Sparse

Integer Factorization: NullSpace, over $\mathbb{Z}/2\mathbb{Z}$, Sparse

Algebraic Attacks: Echelon, LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 20$ bits, Sparse & Dense

List decoding of RS codes: Lattice reduction, over GF$(q)[X]$, Structured

Need for high performance.
The scope of this presentation:

- not an exhaustive overview on linear algebra algorithmic and complexity improvements
- a few guidelines, for the use and design of exact linear algebra in practice
- bridging the theoretical algorithmic development and practical efficiency concerns
Outline

1. Choosing the underlying arithmetic
   - Using boolean arithmetic
   - Using machine word arithmetic
   - Larger field sizes

2. Reductions and building blocks
   - In dense linear algebra
   - In blackbox linear algebra

3. Size dimension trade-offs
   - Hermite normal form
   - Frobenius normal form

4. Parallel exact linear algebra
   - Ingredients for the parallelization
   - Parallel dense linear algebra mod $p$
Choosing the underlying arithmetic

1. Choosing the underlying arithmetic
   - Using boolean arithmetic
   - Using machine word arithmetic
   - Larger field sizes

2. Reductions and building blocks
   - In dense linear algebra
   - In blackbox linear algebra

3. Size dimension trade-offs
   - Hermite normal form
   - Frobenius normal form

4. Parallel exact linear algebra
   - Ingredients for the parallelization
   - Parallel dense linear algebra mod $p$
Achieving high practical efficiency

Most of linear algebra operations boil down to (a lot of)

\[ y \leftarrow y \pm a \times b \]

- dot-product
- matrix-matrix multiplication
- rank 1 update in Gaussian elimination
- Schur complements, …

Efficiency relies on
- fast arithmetic
- fast memory accesses

Here: focus on dense linear algebra
Choosing the underlying arithmetic

Which computer arithmetic?

Many base fields/rings to support

<table>
<thead>
<tr>
<th>Field/Ring</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{Z}_2 )</td>
<td>1 bit</td>
</tr>
<tr>
<td>( \mathbb{Z}_{3,5,7} )</td>
<td>2-3 bits</td>
</tr>
<tr>
<td>( \mathbb{Z}_p )</td>
<td>4-26 bits</td>
</tr>
<tr>
<td>( \mathbb{Z}, \mathbb{Q} )</td>
<td>&gt; 32 bits</td>
</tr>
<tr>
<td>( \mathbb{Z}_p )</td>
<td>&gt; 32 bits</td>
</tr>
</tbody>
</table>
Choosing the underlying arithmetic

**Which computer arithmetic?**

Many base fields/rings to support

<table>
<thead>
<tr>
<th>Field/Ring</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mathbb{Z}_2)</td>
<td>1 bit</td>
</tr>
<tr>
<td>(\mathbb{Z}_3, 5, 7)</td>
<td>2-3 bits</td>
</tr>
<tr>
<td>(\mathbb{Z}_p)</td>
<td>4-26 bits</td>
</tr>
<tr>
<td>(\mathbb{Z}, \mathbb{Q})</td>
<td>&gt; 32 bits</td>
</tr>
<tr>
<td>(\mathbb{Z}_p)</td>
<td>&gt; 32 bits</td>
</tr>
</tbody>
</table>

Available CPU arithmetic

- boolean
- integer (fixed size)
- floating point
- .. and their vectorization
Choosing the underlying arithmetic

Which computer arithmetic?

Many base fields/rings to support

\[
\begin{array}{|c|c|c|}
\hline
\text{Ring} & \text{Bits} & \text{Arithmetic} \\
\hline
\mathbb{Z}_2 & 1 \text{ bit} & \rightsquigarrow \text{bit-packing} \\
\mathbb{Z}_{3,5,7} & 2-3 \text{ bits} & \rightsquigarrow \text{bit-slicing, bit-packing} \\
\mathbb{Z}_p & 4-26 \text{ bits} & \rightsquigarrow \text{CPU arithmetic} \\
\mathbb{Z}, \mathbb{Q} & > 32 \text{ bits} & \rightsquigarrow \text{multiprec. ints, big ints, CRT, lifting} \\
\mathbb{Z}_p & > 32 \text{ bits} & \rightsquigarrow \text{multiprec. ints, big ints, CRT} \\
\hline
\end{array}
\]

Available CPU arithmetic

- boolean
- integer (fixed size)
- floating point
- .. and their vectorization
Choosing the underlying arithmetic

Which computer arithmetic?

Many base fields/rings to support

\[
\begin{align*}
\mathbb{Z}_2 & : 1 \text{ bit} \quad \rightsquigarrow \text{bit-packing} \\
\mathbb{Z}_{3,5,7} & : 2-3 \text{ bits} \quad \rightsquigarrow \text{bit-slicing, bit-packing} \\
\mathbb{Z}_p & : 4-26 \text{ bits} \quad \rightsquigarrow \text{CPU arithmetic} \\
\mathbb{Z}, \mathbb{Q} & > 32 \text{ bits} \quad \rightsquigarrow \text{multiprec. ints, big ints, CRT, lifting} \\
\mathbb{Z}_p & > 32 \text{ bits} \quad \rightsquigarrow \text{multiprec. ints, big ints, CRT} \\
\text{GF}(p^k) & \equiv \mathbb{Z}_p[X]/(Q) \quad \rightsquigarrow \text{Polynomial, Kronecker, Zech log, ...}
\end{align*}
\]

Available CPU arithmetic

- boolean
- integer (fixed size)
- floating point
- .. and their vectorization
Dense linear algebra over $\mathbb{Z}_2$: bit-packing

$$\text{uint64}_t \equiv (\mathbb{Z}_2)^{64} \implies \wedge: \text{bit-wise XOR, (} + \mod 2 \text{)}$$
$$\&: \text{bit-wise AND, (}* \mod 2\text{)}$$

dot-product $(a,b)$

```c
uint64_t t = 0;
for (int k=0; k < N/64; ++k)
    t ^= a[k] & b[k];
c = parity(t)
```

parity($x$)

```c
if (size(x) == 1)
    return x;
else return parity (High(x) ^ Low(x))
```

$$\implies \text{Can be parallelized on 64 instances.}$$
Tabulation:
- avoid computing parities
- balance computation vs communication
- (slight) complexity improvement possible
Tabulation:

- avoid computing parities
- balance computation vs communication
- (slight) complexity improvement possible

The Four Russian method [Arlazarov, Dinic, Kronrod, Faradzev 70]

1. compute all $2^k$ linear combinations of $k$ rows of $B$.

   **Gray code**: each new line costs 1 vector add (vs $k/2$)

2. multiply chunks of length $k$ of $A$ by table look-up

For $k = \log n \Rightarrow O\left(\frac{n^3}{\log n}\right)$.

In practice: choice of $k$ s.t. the table fits in L2 cache.
Choosing the underlying arithmetic

Using boolean arithmetic

Tabulation:

- avoid computing parities
- balance computation vs communication
- (slight) complexity improvement possible

The Four Russian method [Arlazarov, Dinic, Kronrod, Faradzev 70]

1. compute all $2^k$ linear combinations of $k$ rows of $B$.
   
   Gray code: each new line costs 1 vector add (vs $k/2$)

2. multiply chunks of length $k$ of $A$ by table look-up

- For $k = \log n \leadsto O(n^3 / \log n)$.
- In practice: choice of $k$ s.t. the table fits in L2 cache.
Dense linear algebra over $\mathbb{Z}_2$

The M4RI library [Albrecht Bard Hart 10]

- bit-packing
- Method of the Four Russians
- SIMD vectorization of boolean instructions (128 bits registers)
- Cache optimization
- Strassen’s $O(n^{2.81})$ algorithm

<table>
<thead>
<tr>
<th>n</th>
<th>7000</th>
<th>14 000</th>
<th>28 000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD bool arithmetic</td>
<td>2.109s</td>
<td>15.383s</td>
<td>111.82s</td>
</tr>
<tr>
<td>SIMD + 4 Russians</td>
<td>0.256s</td>
<td>2.829s</td>
<td>29.28s</td>
</tr>
<tr>
<td>SIMD + 4 Russians + Strassen</td>
<td>0.257s</td>
<td>2.001s</td>
<td>15.73s</td>
</tr>
</tbody>
</table>

Table: Matrix product $n \times n$ by $n \times n$, on an i5 SandyBridge 2.6Ghz.
Choosing the underlying arithmetic

Using boolean arithmetic

Dense linear algebra over $\mathbb{Z}_3$, $\mathbb{Z}_5$ [Boothby & Bradshaw 09]

\[ \mathbb{Z}_3 = \{0, 1, -1\} = \{00, 01, 10\} \]
Dense linear algebra over $\mathbb{Z}_3, \mathbb{Z}_5$ [Boothby & Bradshaw 09]

$\mathbb{Z}_3 = \{0, 1, -1\} = \{00, 01, 10\} \leadsto \text{add/sub in 7 bool ops}$
Dense linear algebra over $\mathbb{Z}_3, \mathbb{Z}_5$ [Boothby & Bradshaw 09]

$\mathbb{Z}_3 = \{0, 1, -1\} = \{00, 01, 10\} \leadsto \text{add/sub in 7 bool ops}$
$= \{00, 10, 11\} \leadsto \text{add/sub in 6 bool ops}$
Dense linear algebra over \( \mathbb{Z}_3, \mathbb{Z}_5 \) [Boothby & Bradshaw 09]

\[
\mathbb{Z}_3 = \{0, 1, -1\} = \{00, 01, 10\} \rightsquigarrow \text{add/sub in 7 bool ops} \\
= \{00, 10, 11\} \rightsquigarrow \text{add/sub in 6 bool ops}
\]

Bit-slicing

\[
(-1, 0, 1, 0, 1, -1, -1, 0) \in \mathbb{Z}_3^8 \rightarrow (11, 00, 10, 00, 10, 11, 00)
\]

Stored as 2 words

\[
(1, 0, 1, 0, 1, 1, 0) \\
(1, 0, 0, 0, 0, 1, 0)
\]
Dense linear algebra over \( \mathbb{Z}_3, \mathbb{Z}_5 \) [Boothby & Bradshaw 09]

\( \mathbb{Z}_3 = \{0, 1, -1\} = \{00, 01, 10\} \leadsto \text{add/sub in 7 bool ops} \)

\( \mathbb{Z}_3 = \{00, 10, 11\} \leadsto \text{add/sub in 6 bool ops} \)

**Bit-slicing**

\((−1, 0, 1, 0, 1, −1, −1, 0) \in \mathbb{Z}_3^8 \rightarrow (11, 00, 10, 00, 10, 11, 00)\)

Stored as 2 words

\[
\begin{align*}
(1, 0, 1, 0, 1, 1, 0) \\
(1, 0, 0, 0, 0, 1, 0)
\end{align*}
\]

\(\leadsto \vec{y} \leftarrow \vec{y} + x\vec{b} \text{ for } x \in \mathbb{Z}_3, \vec{y}, \vec{b} \in \mathbb{Z}_3^{64} \text{ in 6 boolean word ops.}\)
Dense linear algebra over $\mathbb{Z}_3, \mathbb{Z}_5$ [Boothby & Bradshaw 09]

$\mathbb{Z}_3 = \{0, 1, -1\} = \{00, 01, 10\} \leadsto \text{add/sub in 7 bool ops}$

$= \{00, 10, 11\} \leadsto \text{add/sub in 6 bool ops}$

Bit-slicing

$(-1, 0, 1, 0, 1, -1, -1, 0) \in \mathbb{Z}_3^8 \rightarrow (11, 00, 10, 00, 10, 11, 00)$

Stored as 2 words

\[
\begin{align*}
(1, 0, 1, 0, 1, 1, 0) \\
(1, 0, 0, 0, 0, 1, 0)
\end{align*}
\]

$\leadsto \vec{y} \leftarrow \vec{y} + x\vec{b}$ for $x \in \mathbb{Z}_3$, $\vec{y}, \vec{b} \in \mathbb{Z}_3^{64}$ in 6 boolean word ops.

Recipe for $\mathbb{Z}_5$

- Use redundant representations on 3 bits + bit-slicing
- integer add + bool operations
- Pseudo-reduction mod 5 ($4 \rightarrow 3$ bits) in 8 bool ops found by computer assisted search.
Dense linear algebra over $\mathbb{Z}_p$ for word-size $p$

Delayed modular reductions

1. Compute using integer arithmetic
2. Reduce modulo $p$ only when necessary
Dense linear algebra over $\mathbb{Z}_p$ for word-size $p$

Delayed modular reductions

1. Compute using integer arithmetic
2. Reduce modulo $p$ only when necessary

When to reduce?

Bound the values of all intermediate computations.

- A priori:
  
<table>
<thead>
<tr>
<th>Representation of $\mathbb{Z}_p$</th>
<th>${0 \ldots p - 1}$</th>
<th>${-\frac{p-1}{2} \ldots \frac{p-1}{2}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar product, Classic MatMul</td>
<td>$n(p - 1)^2$</td>
<td>$n\left(\frac{p-1}{2}\right)^2$</td>
</tr>
</tbody>
</table>
Dense linear algebra over $\mathbb{Z}_p$ for word-size $p$

Delayed modular reductions

1. Compute using integer arithmetic
2. Reduce modulo $p$ only when necessary

When to reduce?

Bound the values of all intermediate computations.

- A priori:

<table>
<thead>
<tr>
<th>Representation of $\mathbb{Z}_p$</th>
<th>${0 \ldots p - 1}$</th>
<th>${-\frac{p-1}{2} \ldots \frac{p-1}{2}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar product, Classic MatMul</td>
<td>$n(p - 1)^2$</td>
<td>$n \left(\frac{p-1}{2}\right)^2$</td>
</tr>
<tr>
<td>Strassen-Winograd MatMul ($\ell$ rec. levels)</td>
<td>$(\frac{1+3\ell}{2})^2 \left\lfloor \frac{n}{2^\ell} \right\rfloor (p - 1)^2$</td>
<td>$9\ell \left\lfloor \frac{n}{2^\ell} \right\rfloor \left(\frac{p-1}{2}\right)^2$</td>
</tr>
</tbody>
</table>
Choosing the underlying arithmetic

Using machine word arithmetic

Dense linear algebra over $\mathbb{Z}_p$ for word-size $p$

Delayed modular reductions

1. Compute using integer arithmetic
2. Reduce modulo $p$ only when necessary

When to reduce?

Bound the values of all intermediate computations.

- A priori:

  Representation of $\mathbb{Z}_p$
  
  $\{0 \ldots p-1\} \quad \{ -\frac{p-1}{2} \ldots \frac{p-1}{2} \}$

  Scalar product, Classic MatMul
  
  $n(p-1)^2$

  Strassen-Winograd MatMul ($\ell$ rec. levels)

  $\left( \frac{1+3\ell}{2} \right)^2 \left\lfloor \frac{n}{2^\ell} \right\rfloor (p-1)^2$

  $n \left( \frac{p-1}{2} \right)^2$

  $9\ell \left\lfloor \frac{n}{2^\ell} \right\rfloor \left( \frac{p-1}{2} \right)^2$

- Maintain locally a bounding interval on all matrices computed
Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

1. use CPU’s integer arithmetic units
   
   \[ y + a \times b: \text{correct if } |ab + y| < 2^{63} \Rightarrow |a|, |b| < 2^{31} \]
Choosing the underlying arithmetic

Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

1. Use CPU's integer arithmetic units

   $y += a \times b$: correct if $|ab + y| < 2^{63} \Leftrightarrow |a|, |b| < 2^{31}$

   movq (%rax,%rdx,8), %rax
   imulq -56(%rbp), %rax
   addq %rax, %rcx
   movq -80(%rbp), %rax
Choosing the underlying arithmetic

Using machine word arithmetic

Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

1. use CPU’s integer arithmetic units + vectorization

\[ y += a \times b: \text{correct if } |ab + y| < 2^{63} \implies |a|, |b| < 2^{31} \]

\[
\begin{align*}
\text{movq} & \quad (%rax,%rdx,8), %rax \\
\text{imulq} & \quad -56(%rbp), %rax \\
\text{addq} & \quad %rax, %rcx \\
\text{movq} & \quad -80(%rbp), %rax \\
\text{vpmuludq} & \quad %xmm3, %xmm0,%xmm0 \\
\text{vpaddq} & \quad %xmm2,%xmm0,%xmm0 \\
\text{vpsllq} & \quad $32,%xmm0,%xmm0 \\
\end{align*}
\]
How to compute with (machine word size) integers efficiently?

1. **use CPU’s integer arithmetic units + vectorization**

   \[ y += a \times b: \text{correct if } |ab + y| < 2^{63} \sim |a|, |b| < 2^{31} \]

   
   \[
   \begin{align*}
   \text{movq} & \quad (%rax, %rdx, 8), %rax \\
   \text{imulq} & \quad -56(%rbp), %rax \\
   \text{addq} & \quad %rax, %rcx \\
   \text{movq} & \quad -80(%rbp), %rax
   \end{align*}
   \]

   
   \[
   \begin{align*}
   \text{vpmuludq} & \quad %xmm3, %xmm0, %xmm0 \\
   \text{vpaddq} & \quad %xmm2, %xmm0, %xmm0 \\
   \text{vpsllq} & \quad $32, %xmm0, %xmm0$
   \end{align*}
   \]

2. **use CPU’s floating point units**

   \[ y += a \times b: \text{correct if } |ab + y| < 2^{53} \sim |a|, |b| < 2^{26} \]

   
   \[
   \begin{align*}
   \text{movsd} & \quad (%rax, %rdx, 8), %xmm0 \\
   \text{mulsd} & \quad -56(%rbp), %xmm0 \\
   \text{addsd} & \quad %xmm0, %xmm1 \\
   \text{movq} & \quad %xmm1, %rax \\
   \text{vinsertf128} & \quad $0x1, 16(%rcx, %rax), %ymm0, %ymm0 \\
   \text{vmulpd} & \quad %ymm1, %ymm0, %ymm0 \\
   \text{vaddpd} & \quad (%rsi, %rax), %ymm0, %ymm0 \\
   \text{vmovapd} & \quad %ymm0, (%rsi, %rax)
   \end{align*}
   \]
Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

1. use CPU’s **integer arithmetic units** + vectorization
   \[ y += a \times b: \text{correct if } |ab + y| < 2^{63} \quad \sim \quad |a|, |b| < 2^{31} \]
   
   - `movq (%rax, %rdx, 8), %rax`
   - `imulq -56(%rbp), %rax`
   - `addq %rax, %rcx`
   - `movq -80(%rbp), %rax`

2. use CPU’s **floating point units**
   \[ y += a \times b: \text{correct if } |ab + y| < 2^{53} \quad \sim \quad |a|, |b| < 2^{26} \]
   
   - `movsd (%rax, %rdx, 8), %xmm0`
   - `mulsd -56(%rbp), %xmm0`
   - `addsd %xmm0, %xmm1`
   - `movq %xmm1, %rax`
Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

1. **use CPU’s integer arithmetic units + vectorization**
   
   \[ y += a \times b: \text{correct if } |ab + y| < 2^{63} \iff |a|, |b| < 2^{31} \]

   ```
   movq (%rax,%rdx,8), %rax
   imulq -56(%rbp), %rax
   addq %rax, %rcx
   movq -80(%rbp), %rax
   vpmuludq %xmm3, %xmm0,%xmm0
   vpaddq %xmm2,%xmm0,%xmm0
   vpsllq $32,%xmm0,%xmm0
   ```

2. **use CPU’s floating point units + vectorization**

   \[ y += a \times b: \text{correct if } |ab + y| < 2^{53} \iff |a|, |b| < 2^{26} \]

   ```
   movsd (%rax,%rdx,8), %xmm0
   mulsd -56(%rbp), %xmm0
   addsd %xmm0, %xmm1
   movq %xmm1, %rax
   vinsertf128 $0x1, 16(%rcx,%rax), %ymm0,
   vmulpd %ymm1, %ymm0, %ymm0
   vaddpd (%rsi,%rax),%ymm0, %ymm0
   vmovapd %ymm0, (%rsi,%rax)
   ```
Exploiting *in-core* parallelism

**Ingredients**

**SIMD**: Single Instruction Multiple Data:  
1 arith. unit acting on a vector of data

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>64 bits</td>
<td>SSE</td>
<td>128 bits</td>
<td>AVX</td>
<td>256 bits</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\end{align*}
\]
Exploiting *in-core* parallelism

**Ingredients**

**SIMD:** Single Instruction Multiple Data:

1 arith. unit acting on a vector of data

- MMX: 64 bits
- SSE: 128 bits
- AVX: 256 bits
- AVX-512: 512 bits

**Pipeline:** amortize the latency of an operation when used repeatedly

throughput of 1 op/ Cycle for all arithmetic ops considered here
Exploiting *in-core* parallelism

**Ingredients**

**SIMD:** Single Instruction Multiple Data:
1 arith. unit acting on a vector of data

- MMX 64 bits
- SSE 128 bits
- AVX 256 bits
- AVX-512 512 bits

**Pipeline:** amortize the latency of an operation when used repeatedly
throughput of 1 op/ Cycle for all arithmetic ops considered here

**Execution Unit parallelism:** multiple arith. units acting simultaneously on distinct registers
**SIMD and vectorization**

**Intel Sandybridge micro-architecture**

- **Scheduler**
  - Port 0: FMUL, Int MUL
  - Port 1: FADD, Int ADD
  - Port 5: Int ADD

- Performance:
  - 4 x 64 = 256 bits
  - 2 x 64 = 128 bits

- Performs at every clock cycle:
  - 1 Floating Pt. Mul × 4
  - 1 Floating Pt. Add × 4
  - Or:
    - 1 Integer Mul × 2
    - 2 Integer Add × 2
SIMD and vectorization

Intel Haswell micro-architecture

Scheduler

Port 0
- FMA
- Int MUL

Port 1
- FMA
- Int ADD

Port 5
- Int ADD

4 x 64 = 256 bits

Performs at every clock cycle:
- 1 Floating Pt. Mul & Add \( \times 4 \)
- 1 Floating Pt. Add & Add \( \times 4 \)

Or:
- 1 Integer Mul \( \times 4 \)
- 2 Integer Add \( \times 4 \)

FMA: Fused Multiplying & Accumulate, \( c += a \cdot b \)
SIMD and vectorization

AMD Bulldozer micro-architecture

Performs at every clock cycle:

- 2 Floating Pt. Mul & Add × 2

Or:

- 1 Integer Mul × 2
- 2 Integer Add × 2

FMA: Fused Multiplying & Accumulate, $c += a \times b$
Choosing the underlying arithmetic

Using machine word arithmetic

SIMD and vectorization

Intel Nehalem micro-architecture

Scheduler

Port 0

\[ 2 \times 64 = 128 \text{ bits} \]

\[ \text{F_MUL} \]

\[ \text{Int MUL} \]

Port 1

\[ \text{FADD} \]

\[ \text{Int ADD} \]

Port 5

\[ \text{Int ADD} \]

\[ 2 \times 64 = 128 \text{ bits} \]

Performs at every clock cycle:

\[ \begin{align*}
\text{1 Floating Pt. Mul} \times 2 \\
\text{1 Floating Pt. Add} \times 2
\end{align*} \]

Or:

\[ \begin{align*}
\text{1 Integer Mul} \times 2 \\
\text{2 Integer Add} \times 2
\end{align*} \]
### Summary: 64 bits AXPY throughput

<table>
<thead>
<tr>
<th></th>
<th>Register size</th>
<th># Adders</th>
<th># Multipliers</th>
<th># FMA</th>
<th># daxpy / Cycle</th>
<th>CPU Freq. (Ghz)</th>
<th>Speed of Light (Gfops)</th>
<th>Speed in practice (Gfops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Haswell</td>
<td>INT 256</td>
<td>2</td>
<td>1</td>
<td></td>
<td>4</td>
<td>3.5</td>
<td>28</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Sandybridge</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>3.5</td>
<td>13.2</td>
<td>12.1</td>
</tr>
<tr>
<td></td>
<td>FP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD Bulldozer</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2.1</td>
<td>8.4</td>
<td>6.44</td>
</tr>
<tr>
<td></td>
<td>FP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Nehalem</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2.66</td>
<td>10.6</td>
<td>4.47</td>
</tr>
<tr>
<td></td>
<td>FP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSE4</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2.66</td>
<td>10.6</td>
<td>9.6</td>
</tr>
<tr>
<td></td>
<td>FP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD K10</td>
<td>INT 64</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2.4</td>
<td>4.8</td>
<td>4.8</td>
</tr>
<tr>
<td></td>
<td>SSE4a FP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Speed of light:** \( \text{CPU freq} \times (\text{# daxpy / Cycle}) \times 2 \)
### Summary: 64 bits AXPY throughput

<table>
<thead>
<tr>
<th>Processor</th>
<th>Register size</th>
<th># Adders</th>
<th># Multipliers</th>
<th># FMA</th>
<th># daxpy/Cycle</th>
<th>CPU Freq. (Ghz)</th>
<th>Speed of Light (Gfops)</th>
<th>Speed in practice (Gfops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Haswell AVX2</td>
<td>256</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3.5</td>
<td>28</td>
<td>23.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.5</td>
<td>56</td>
<td>49.2</td>
</tr>
<tr>
<td>Intel Sandybridge AVX1</td>
<td>128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3.3</td>
<td>13.2</td>
<td>12.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>256</td>
<td></td>
<td></td>
<td>8</td>
<td>3.5</td>
<td>26.4</td>
<td>24.6</td>
<td></td>
</tr>
<tr>
<td>AMD Bulldozer FMA4</td>
<td>64</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2.1</td>
<td>8.4</td>
<td>6.44</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td></td>
<td></td>
<td>4</td>
<td>2.1</td>
<td>16.8</td>
<td>13.1</td>
<td></td>
</tr>
<tr>
<td>Intel Nehalem SSE4</td>
<td>128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2.66</td>
<td>10.6</td>
<td>4.47</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td></td>
<td></td>
<td>2</td>
<td>2.66</td>
<td>10.6</td>
<td>9.6</td>
<td></td>
</tr>
<tr>
<td>AMD K10 SSE4a</td>
<td>64</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2.4</td>
<td>4.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td></td>
<td></td>
<td>4</td>
<td>2.4</td>
<td>9.6</td>
<td>8.93</td>
<td></td>
</tr>
</tbody>
</table>

**Speed of light:** $\text{CPU freq} \times (\# \text{daxpy} / \text{Cycle}) \times 2$
## Summary: 64 bits AXPY throughput

<table>
<thead>
<tr>
<th>CPU Family</th>
<th>Register size</th>
<th># Adders</th>
<th># Multipliers</th>
<th># FMA</th>
<th>daxpy / Cycle</th>
<th>CPU Freq. (Ghz)</th>
<th>Speed of Light (Gfops)</th>
<th>Speed in practice (Gfops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Haswell AVX2</td>
<td>INT 256</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3.5</td>
<td>28</td>
<td>23.3</td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>3.5</td>
<td>56</td>
<td>49.2</td>
</tr>
<tr>
<td>Intel Sandybridge AVX1</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>3.3</td>
<td>13.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td>1</td>
<td>1</td>
<td></td>
<td>4</td>
<td>3.3</td>
<td>26.4</td>
<td></td>
</tr>
<tr>
<td>AMD Bulldozer FMA4</td>
<td>INT 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.1</td>
<td>8.4</td>
<td>6.44</td>
</tr>
<tr>
<td></td>
<td>FP 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.1</td>
<td>16.8</td>
<td>13.1</td>
</tr>
<tr>
<td>Intel Nehalem SSE4</td>
<td>INT 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.66</td>
<td>10.6</td>
<td>4.47</td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.66</td>
<td>10.6</td>
<td>9.6</td>
</tr>
<tr>
<td>AMD K10 SSE4a</td>
<td>INT 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.4</td>
<td>4.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.4</td>
<td>9.6</td>
<td>8.93</td>
</tr>
</tbody>
</table>

**Speed of light:** CPU freq × ( # daxpy / Cycle) × 2
## Summary: 64 bits AXPY throughput

<table>
<thead>
<tr>
<th></th>
<th>Register size</th>
<th># Adders</th>
<th># Multipliers</th>
<th># FMA</th>
<th># daxpy / Cycle</th>
<th>CPU Freq. (Ghz)</th>
<th>Speed of Light (Gfops)</th>
<th>Speed in practice (Gfops)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intel Haswell</strong></td>
<td>INT 256</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3.5</td>
<td>28</td>
<td>23.3</td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AVX2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Intel Sandybridge</strong></td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>3.3</td>
<td>13.2</td>
<td>12.1</td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AVX1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AMD Bulldozer</strong></td>
<td>INT 128</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
<td>3.3</td>
<td>26.4</td>
<td>24.6</td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FMA4</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Intel Nehalem</strong></td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td></td>
<td>2.66</td>
<td>10.6</td>
<td>4.47</td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SSE4</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AMD K10</strong></td>
<td>INT 128</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
<td>2.4</td>
<td>4.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SSE4a</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Speed of light:** \( \text{CPU freq} \times (\# \text{daxpy} / \text{Cycle}) \times 2 \)
### Summary: 64 bits AXPY throughput

<table>
<thead>
<tr>
<th></th>
<th>Register size</th>
<th># Adders</th>
<th># Multipliers</th>
<th># FMA</th>
<th>daxpy/Cycle</th>
<th>CPU Freq. (Ghz)</th>
<th>Speed of Light (Gfops)</th>
<th>Speed in practice (Gfops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Haswell INT</td>
<td>256</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3.5</td>
<td>28</td>
<td>23.3</td>
</tr>
<tr>
<td>AVX2 FP</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>56</td>
<td>49.2</td>
</tr>
<tr>
<td>Intel Sandybridge INT</td>
<td>128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3.3</td>
<td>13.2</td>
<td>12.1</td>
</tr>
<tr>
<td>AVX1 FP</td>
<td>256</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>26.4</td>
<td>24.6</td>
</tr>
<tr>
<td>AMD Bulldozer INT</td>
<td>128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2.1</td>
<td>8.4</td>
<td></td>
</tr>
<tr>
<td>FMA4 FP</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Nehalem INT</td>
<td>128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2.1</td>
<td>8.4</td>
<td></td>
</tr>
<tr>
<td>SSE4 FP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD K10 INT</td>
<td>128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2.1</td>
<td>8.4</td>
<td></td>
</tr>
<tr>
<td>SSE4a FP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Speed of light:** CPU freq × ( # daxpy / Cycle) × 2
## Summary: 64 bits AXPY throughput

<table>
<thead>
<tr>
<th>CPU</th>
<th>Register size</th>
<th># Adders</th>
<th># Multipliers</th>
<th># FMA</th>
<th># daxpy / Cycle</th>
<th>CPU Freq. (Ghz)</th>
<th>Speed of Light (Gfops)</th>
<th>Speed in practice (Gfops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Haswell AVX2</td>
<td>INT 256 2 1 4 3.5 28</td>
<td>256 2 1 4 3.5 28 23.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 256 2 1 4 3.5 56</td>
<td>256 2 1 4 3.5 56 49.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Sandybridge AVX1</td>
<td>INT 128 2 1 2 13.2</td>
<td>128 2 1 2 13.2 12.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 256 1 1 4 26.4</td>
<td>256 1 1 4 26.4 24.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD Bulldozer FMA4</td>
<td>INT 128 2 1 2 8.4</td>
<td>128 2 1 2 8.4 6.44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 128 2 4 2.1 16.8</td>
<td>128 2 4 2.1 16.8 13.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Nehalem SSE4</td>
<td>INT 2 4 2.1 16.8</td>
<td>128 2 4 2.1 16.8 13.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 2 4 2.1 16.8</td>
<td>256 2 4 2.1 16.8 24.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD K10 SSE4a</td>
<td>INT 2 4 2.1 16.8</td>
<td>256 2 4 2.1 16.8 24.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 2 4 2.1 16.8</td>
<td>256 2 4 2.1 16.8 24.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Speed of light:** CPU freq × ( # daxpy / Cycle) × 2
### Summary: 64 bits AXPY throughput

<table>
<thead>
<tr>
<th></th>
<th>Register size</th>
<th># Adders</th>
<th># Multipliers</th>
<th># FMA</th>
<th># daxpy/Cycle</th>
<th>CPU Freq. (Ghz)</th>
<th>Speed of Light (Gfops)</th>
<th>Speed in practice (Gfops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Haswell</td>
<td>INT 256</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td></td>
<td>3.5</td>
<td>28</td>
<td>23.3</td>
</tr>
<tr>
<td>AVX2</td>
<td>FP 256</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>3.5</td>
<td>56</td>
<td>49.2</td>
</tr>
<tr>
<td>Intel Sandybridge</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>3.3</td>
<td>13.2</td>
<td>12.1</td>
</tr>
<tr>
<td>AVX1</td>
<td>FP 256</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
<td>3.3</td>
<td>26.4</td>
<td>24.6</td>
</tr>
<tr>
<td>AMD Bulldozer</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>2.1</td>
<td>8.4</td>
<td>6.44</td>
</tr>
<tr>
<td>FMA4</td>
<td>FP 128</td>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>2.1</td>
<td>16.8</td>
<td>13.1</td>
</tr>
<tr>
<td>Intel Nehalem</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>2.66</td>
<td>10.6</td>
<td></td>
</tr>
<tr>
<td>SSE4</td>
<td>FP 128</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td>2.66</td>
<td>10.6</td>
<td></td>
</tr>
<tr>
<td>AMD K10</td>
<td>INT 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSE4a</td>
<td>FP 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Speed of light:** \( \text{CPU freq} \times (\# \text{daxpy} / \text{Cycle}) \times 2 \)
# Summary: 64 bits AXPY throughput

<table>
<thead>
<tr>
<th></th>
<th>Register size</th>
<th># Adders</th>
<th># Multipliers</th>
<th># FMA</th>
<th>daxpy / Cycle</th>
<th>CPU Freq. (Ghz)</th>
<th>Speed of Light (Gfops)</th>
<th>Speed in practice (Gfops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Haswell</td>
<td>INT</td>
<td>256</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3.5</td>
<td>28</td>
<td>23.3</td>
</tr>
<tr>
<td></td>
<td>FP</td>
<td>256</td>
<td></td>
<td></td>
<td>2</td>
<td>3.5</td>
<td>56</td>
<td>49.2</td>
</tr>
<tr>
<td>Intel Sandybridge</td>
<td>INT</td>
<td>128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3.3</td>
<td>13.2</td>
<td>12.1</td>
</tr>
<tr>
<td></td>
<td>FP</td>
<td>256</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>3.3</td>
<td>26.4</td>
<td>24.6</td>
</tr>
<tr>
<td>AMD Bulldozer</td>
<td>INT</td>
<td>128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2.1</td>
<td>8.4</td>
<td>6.44</td>
</tr>
<tr>
<td></td>
<td>FP</td>
<td>128</td>
<td></td>
<td></td>
<td>2</td>
<td>2.1</td>
<td>16.8</td>
<td>13.1</td>
</tr>
<tr>
<td>Intel Nehalem</td>
<td>INT</td>
<td>128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2.66</td>
<td>10.6</td>
<td>4.47</td>
</tr>
<tr>
<td></td>
<td>FP</td>
<td>128</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2.66</td>
<td>10.6</td>
<td>9.6</td>
</tr>
<tr>
<td>AMD K10</td>
<td>INT</td>
<td>128</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2.66</td>
<td>10.6</td>
<td>9.6</td>
</tr>
<tr>
<td></td>
<td>FP</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Speed of light:** CPU freq × ( # daxpy / Cycle) ×2
### Summary: 64 bits AXPY throughput

<table>
<thead>
<tr>
<th></th>
<th>Register size</th>
<th># Adders</th>
<th># Multipliers</th>
<th># FMA</th>
<th># daxpy / Cycle</th>
<th>CPU Freq. (Ghz)</th>
<th>Speed of Light (Gfops)</th>
<th>Speed in practice (Gfops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Haswell</td>
<td>INT 256</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3.5</td>
<td>28</td>
<td>23.3</td>
<td>24.6</td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td>2</td>
<td>8</td>
<td></td>
<td>3.5</td>
<td>56</td>
<td>49.2</td>
<td></td>
</tr>
<tr>
<td>Intel Sandybridge</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3.3</td>
<td>13.2</td>
<td>12.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 256</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>3.3</td>
<td>26.4</td>
<td>24.6</td>
<td></td>
</tr>
<tr>
<td>AMD Bulldozer</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2.1</td>
<td>8.4</td>
<td>6.44</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 128</td>
<td>2</td>
<td></td>
<td>4</td>
<td>2.1</td>
<td>16.8</td>
<td>13.1</td>
<td></td>
</tr>
<tr>
<td>Intel Nehalem</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2.66</td>
<td>10.6</td>
<td>4.47</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 128</td>
<td>1</td>
<td></td>
<td>2</td>
<td>2.66</td>
<td>10.6</td>
<td>9.6</td>
<td></td>
</tr>
<tr>
<td>AMD K10</td>
<td>INT 64</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2.4</td>
<td>4.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP 128</td>
<td>1</td>
<td></td>
<td>2</td>
<td>2.4</td>
<td>9.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Speed of light:** CPU freq × ( # daxpy / Cycle) × 2
## Summary: 64 bits AXPY throughput

<table>
<thead>
<tr>
<th></th>
<th>Register size</th>
<th># Adders</th>
<th># Multipliers</th>
<th># FMA</th>
<th># daxpy / Cycle</th>
<th>CPU Freq. (Ghz)</th>
<th>Speed of Light (Gfops)</th>
<th>Speed in practice (Gfops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Haswell</td>
<td>INT 256</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3.5</td>
<td>28</td>
<td>23.3</td>
<td>49.2</td>
</tr>
<tr>
<td>AVX2</td>
<td>FP 256</td>
<td></td>
<td>2</td>
<td>8</td>
<td>3.5</td>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Sandybridge</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3.3</td>
<td>13.2</td>
<td>12.1</td>
<td></td>
</tr>
<tr>
<td>AVX1</td>
<td>FP 256</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>3.3</td>
<td>26.4</td>
<td>24.6</td>
<td></td>
</tr>
<tr>
<td>AMD Bulldozer</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2.1</td>
<td>8.4</td>
<td>6.44</td>
<td></td>
</tr>
<tr>
<td>FMA4</td>
<td>FP 128</td>
<td></td>
<td>2</td>
<td>4</td>
<td>2.1</td>
<td>16.8</td>
<td>13.1</td>
<td></td>
</tr>
<tr>
<td>Intel Nehalem</td>
<td>INT 128</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2.66</td>
<td>10.6</td>
<td>4.47</td>
<td></td>
</tr>
<tr>
<td>SSE4</td>
<td>FP 128</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2.66</td>
<td>10.6</td>
<td>9.6</td>
<td></td>
</tr>
<tr>
<td>AMD K10</td>
<td>INT 64</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2.4</td>
<td>4.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSE4a</td>
<td>FP 128</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2.4</td>
<td>9.6</td>
<td>8.93</td>
<td></td>
</tr>
</tbody>
</table>

**Speed of light:** \( \text{CPU freq} \times (\ # \ daxpy / \ Cycle) \times 2 \)
Computing over fixed size integers: ressources

Micro-architecture bible: Agner Fog’s software optimization resources [www.agner.org/optimize]

Experiments:

dgemm (double): OpenBLAS [http://www.openblas.net/]

igemm (int64_t): Eigen [http://eigen.tuxfamily.org/] & FFLAS-FFPACK [linalg.org/projects/fflas-ffpack]
### Integer Packing

#### 32 bits: half the precision twice the speed

<table>
<thead>
<tr>
<th></th>
<th>double</th>
<th>float</th>
<th>double</th>
<th>float</th>
<th>double</th>
<th>float</th>
<th>double</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gfops</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel SandyBridge</td>
<td>24.7</td>
<td>49.1</td>
<td>12.1</td>
<td>24.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Haswell</td>
<td>49.2</td>
<td>77.6</td>
<td>23.3</td>
<td>27.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD Bulldozer</td>
<td>13.0</td>
<td>20.7</td>
<td>6.63</td>
<td>11.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computing over fixed size integers

SandyBridge i5-3320M@3.3Ghz. \( n = 2000 \).

**Take home message**

- Floating pt. arith. delivers the highest speed (except in corner cases)
- 32 bits twice as fast as 64 bits
Choosing the underlying arithmetic

Computing over fixed size integers

SandyBridge i5-3320M@3.3Ghz. \( n = 2000 \).

Take home message

- Floating pt. arith. delivers the highest speed (except in corner cases)
- 32 bits twice as fast as 64 bits
- best bit computation throughput for double precision floating points.
Larger finite fields: $\log_2 p \geq 32$

As before:
1. Use adequate integer arithmetic
2. reduce modulo $p$ only when necessary

Which integer arithmetic?
1. big integers (GMP)
2. fixed size multiprecision (Givaro-RecInt)
3. Residue Number Systems (Chinese Remainder theorem)
   $\mapsto$ using moduli delivering optimum bitspeed
Larger finite fields: $\log_2 p \geq 32$

As before:
1. Use adequate integer arithmetic
2. reduce modulo $p$ only when necessary

Which integer arithmetic?
1. big integers (GMP)
2. fixed size multiprecision (Givaro-RecInt)
3. Residue Number Systems (Chinese Remainder theorem)
   $\leadsto$ using moduli delivering optimum bitspeed

<table>
<thead>
<tr>
<th>$\log_2 p$</th>
<th>50</th>
<th>100</th>
<th>150</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMP</td>
<td>58.1s</td>
<td>94.6s</td>
<td>140s</td>
</tr>
<tr>
<td>RecInt</td>
<td>5.7s</td>
<td>28.6s</td>
<td>837s</td>
</tr>
<tr>
<td>RNS</td>
<td>0.785s</td>
<td>1.42s</td>
<td>1.88s</td>
</tr>
</tbody>
</table>

$n = 1000$, on an Intel SandyBridge.
Choosing the underlying arithmetic

In practice

fgemm $C = A \times B$ $n = 2000$

- float mod $p$
- double mod $p$
- int64 mod $p$
- RNS mod $p$

Speed (Gfops)

fgemm $C = A \times B$ $n = 2000$

- float mod $p$
- double mod $p$
- int64 mod $p$
- RNS mod $p$

Bit speed (Gbops)
In practice

fgemm $C = A \times B$ $n = 2000$

- float mod $p$
- double mod $p$
- int64 mod $p$
- RNS mod $p$

**Speed (Gflops)**

```
0  5  10  15  20  25  30  35
```

**Modulo p (bitsize)**

```
0  20  40  60  80  100  120  140
```

**Bit speed (Gbops)**

```
0  5  10  15  20  25  30  35
```
In practice

Choosing the underlying arithmetic

Larger field sizes

C. Pernet
Outline

1. Choosing the underlying arithmetic
   - Using boolean arithmetic
   - Using machine word arithmetic
   - Larger field sizes

2. Reductions and building blocks
   - In dense linear algebra
   - In blackbox linear algebra

3. Size dimension trade-offs
   - Hermite normal form
   - Frobenius normal form

4. Parallel exact linear algebra
   - Ingredients for the parallelization
   - Parallel dense linear algebra mod $p$
Huge number of algorithmic variants for a given computation in $O(n^3)$. Need to structure the design of set of routines:

- Focus tuning effort on a single routine
- Some operations deliver better efficiency:
  - in practice: memory access pattern
  - in theory: better algorithms
Memory access pattern

- **The memory wall**: communication speed improves slower than arithmetic
Memory access pattern

- **The memory wall**: communication speed improves slower than arithmetic
- Deep memory hierarchy

![Diagram showing memory hierarchy and CPU arithmetic throughput and memory speed over time.](image-url)
Memory access pattern

- **The memory wall**: communication speed improves slower than arithmetic
- Deep memory hierarchy
- Need to overlap communications by computation

Design of BLAS 3 [Dongarra & Al. 87]

- Group all ops in **Matrix products gemm**: Work $O(n^3) >>$ Data $O(n^2)$

MatMul has become a building block in practice
Sub-cubic linear algebra

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskii, etc)
Sub-cubic linear algebra

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskii, etc)

Matrix Multiplication $\leadsto O(n^\omega)$

- [Strassen 69]: $O(n^{2.807})$
- [Schönhage 81]: $O(n^{2.52})$
- [Coppersmith, Winograd 90]: $O(n^{2.375})$
- [Le Gall 14]: $O(n^{2.3728639})$
Sub-cubic linear algebra

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskii, etc)

Matrix Multiplication $\sim O(n^\omega)$

<table>
<thead>
<tr>
<th>Reference</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Strassen 69]</td>
<td>$O(n^{2.807})$</td>
</tr>
<tr>
<td>[Schönhage 81]</td>
<td>$O(n^{2.52})$</td>
</tr>
<tr>
<td>[Coppersmith, Winograd 90]</td>
<td>$O(n^{2.375})$</td>
</tr>
<tr>
<td>[Le Gall 14]</td>
<td>$O(n^{2.3728639})$</td>
</tr>
</tbody>
</table>

Other operations

<table>
<thead>
<tr>
<th>Reference</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Strassen 69]: Inverse</td>
<td>$O(n^\omega)$</td>
</tr>
<tr>
<td>[Schönhage 72]: QR</td>
<td>$O(n^\omega)$</td>
</tr>
<tr>
<td>[Bunch, Hopcroft 74]: LU</td>
<td>$O(n^\omega)$</td>
</tr>
<tr>
<td>[Ibarra &amp; al. 82]: Rank</td>
<td>$O(n^\omega)$</td>
</tr>
<tr>
<td>[Keller-Gehrig 85]: CharPoly</td>
<td>$O(n^\omega \log n)$</td>
</tr>
</tbody>
</table>
Sub-cubic linear algebra

< 1969: \( O(n^3) \) for everyone (Gauss, Householder, Danilevskii, etc)

Matrix Multiplication \( \sim O(n^\omega) \)

- [Strassen 69]: \( O(n^{2.807}) \)
- [Schönhage 81]: \( O(n^{2.52}) \)
- [Coppersmith, Winograd 90]: \( O(n^{2.375}) \)
- [Le Gall 14]: \( O(n^{2.3728639}) \)

Other operations

- [Strassen 69]: Inverse in \( O(n^\omega) \)
- [Schönhage 72]: QR in \( O(n^\omega) \)
- [Bunch, Hopcroft 74]: LU in \( O(n^\omega) \)
- [Ibarra & al. 82]: Rank in \( O(n^\omega) \)
- [Keller-Gehrig 85]: CharPoly in \( O(n^\omega \log n) \)

MatMul has become a building block in theory theoretical reductions
Reductions: theory

- HNF($\mathbb{Z}$)
- SNF($\mathbb{Z}$)
- LinSys($\mathbb{Z}$)
- MM($\mathbb{Z}$)
- HNF($\mathbb{Z}_p$)
- Det($\mathbb{Z}_p$)
- MinPoly($\mathbb{Z}_p$)
- TRSM($\mathbb{Z}_p$)
- MM($\mathbb{Z}_p$)
- LinSys($\mathbb{Z}_p$)
- LU($\mathbb{Z}_p$)
- CharPoly($\mathbb{Z}_p$)

Road map towards efficiency in practice:
1. Tune the MatMul building block.
2. Tune the reductions.
3. New reductions.

Common mistrust: Fast linear algebra is never faster numerically unstable.
Lucky coincidence: same building block in theory and in practice $\Rightarrow$ reduction trees are still relevant.
Reductions: theory

Common mistrust

Fast linear algebra is

\( \times \) never faster

\( \times \) numerically unstable
Reductions: theory and practice

Common mistrust
Fast linear algebra is
- never faster
- numerically unstable

Lucky coincidence
- same building block in theory and in practice

⇝ reduction trees are still relevant
Reductions: theory and practice

Common mistrust
Fast linear algebra is
× never faster
× numerically unstable

Lucky coincidence
✓ same building block in theory and in practice

⇝ reduction trees are still relevant

Road map towards efficiency in practice
1. Tune the MatMul building block.
2. Tune the reductions.
Putting it together: MatMul building block over $\mathbb{Z}/p\mathbb{Z}$

Ingredients [FFLAS-FFPACK library]

- Compute over $\mathbb{Z}$ and delay modular reductions

$$k \left( \frac{p-1}{2} \right)^2 < 2^{\text{mantissa}}$$
Putting it together: MatMul building block over $\mathbb{Z}/p\mathbb{Z}$

**Ingredients [FFLAS-FFPACK library]**

- Compute over $\mathbb{Z}$ and delay modular reductions
  \[ k \left( \frac{p-1}{2} \right)^2 < 2^{53} \]
- Fastest integer arithmetic: double
- Cache optimizations
  \[ \sim \rightarrow \text{numerical BLAS} \]
Putting it together: MatMul building block over $\mathbb{Z}/p\mathbb{Z}$

**Ingredients [FFLAS-FFPACK library]**

- Compute over $\mathbb{Z}$ and delay modular reductions
  \[ 9^\ell \left\lfloor \frac{k}{2^\ell} \right\rfloor \left( \frac{p-1}{2} \right)^2 < 2^{53} \]

- Fastest integer arithmetic: double

- Cache optimizations

- Strassen-Winograd $6n^{2.807} + \ldots$
Putting it together: MatMul building block over $\mathbb{Z}/p\mathbb{Z}$

**Ingredients [FFLAS-FFPACK library]**

- Compute over $\mathbb{Z}$ and delay modular reductions
  \[ 9^\ell \left\lfloor \frac{k}{2^\ell} \right\rfloor \left( \frac{p-1}{2} \right)^2 < 2^{53} \]

- Fastest integer arithmetic: double

- Cache optimizations

- Strassen-Winograd $6n^{2.807} + \ldots$

with memory efficient schedules [Boyer, Dumas, P. and Zhou 09]

**Tradeoffs:**

- Extra memory
- Overwriting input
- Leading constant
- Fully in-place in $7.2n^{2.807} + \ldots$
Sequential Matrix Multiplication

\[ \text{i5–3320M at 2.6Ghz with AVX 1} \]

\[ 2n^3 / \text{time}/10^9 \text{ (Gfops equiv.)} \]

\[ \text{matrix dimension} \]

\[ p = 83, \Rightarrow 1 \text{ mod } / 10000 \text{ mul.} \]

\[ p = 821, \Rightarrow 1 \text{ mod } / 10 \text{ mul.} \]
Sequential Matrix Multiplication

\[ p = 83, \quad \equiv 1 \mod / 10000 \text{ mul.} \]
Sequential Matrix Multiplication

\[ \text{i}5-3320\text{M at 2.6Ghz with AVX 1} \]

\[ 2n^3/\text{time}/10^9 \text{ (Gfops equiv.)} \]

\[ \text{matrix dimension} \]

\[ p = 83, \sim 1 \mod / 10000 \text{ mul.} \]

\[ p = 821, \sim 1 \mod / 100 \text{ mul.} \]
Sequential Matrix Multiplication

\[ p = 83, \Rightarrow 1 \mod / 10000 \text{ mul.} \]
\[ p = 821, \Rightarrow 1 \mod / 100 \text{ mul.} \]
\[ p = 1898131, \Rightarrow 1 \mod / 10000 \text{ mul.} \]
\[ p = 18981307, \Rightarrow 1 \mod / 100 \text{ mul.} \]
Reductions and building blocks

In dense linear algebra

Reductions in dense linear algebra

LU decomposition

- Block recursive algorithm \( \leadsto \) reduces to MatMul \( \leadsto O(n^\omega) \)

<table>
<thead>
<tr>
<th></th>
<th>( n = 1000 )</th>
<th>( n = 5000 )</th>
<th>( n = 10000 )</th>
<th>( n = 15000 )</th>
<th>( n = 20000 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK-dgetrf</td>
<td>0.024s</td>
<td>2.01s</td>
<td>14.88s</td>
<td>48.78s</td>
<td>113.66s</td>
</tr>
<tr>
<td>fflas-ffpack</td>
<td>0.058s</td>
<td>2.46s</td>
<td>16.08s</td>
<td>47.47s</td>
<td>105.96s</td>
</tr>
</tbody>
</table>

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9
Reductions in dense linear algebra

LU decomposition

- Block recursive algorithm \(\leadsto\) reduces to MatMul \(\leadsto O(n^\omega)\)

<table>
<thead>
<tr>
<th>(n)</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>15000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK-dgetrf</td>
<td>0.024s</td>
<td>2.01s</td>
<td>14.88s</td>
<td>48.78s</td>
<td>113.66s</td>
</tr>
<tr>
<td>fflas-ffpack</td>
<td>0.058s</td>
<td>2.46s</td>
<td>16.08s</td>
<td>47.47s</td>
<td>105.96s</td>
</tr>
</tbody>
</table>

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9

Characteristic Polynomial

- A new reduction to matrix multiplication in \(O(n^\omega)\).

<table>
<thead>
<tr>
<th>(n)</th>
<th>1000</th>
<th>2000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>magma-v2.19-9</td>
<td>1.38s</td>
<td>24.28s</td>
<td>332.7s</td>
<td>2497s</td>
</tr>
<tr>
<td>fflas-ffpack</td>
<td>0.532s</td>
<td>2.936s</td>
<td>32.71s</td>
<td>219.2s</td>
</tr>
</tbody>
</table>

Intel Ivy-Bridge i5-3320 2.6Ghz using OpenBLAS-0.2.9
Reductions and building blocks

## Reductions in dense linear algebra

### LU decomposition

- Block recursive algorithm \( \leadsto \) reduces to MatMul \( \leadsto O(n^\omega) \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>15000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK-dgetrf</td>
<td><strong>0.024s</strong></td>
<td><strong>2.01s</strong></td>
<td><strong>14.88s</strong></td>
<td>48.78s</td>
<td>113.66s</td>
</tr>
<tr>
<td>fflas-ffpack</td>
<td><strong>0.058s</strong></td>
<td><strong>2.46s</strong></td>
<td><strong>16.08s</strong></td>
<td><strong>47.47s</strong></td>
<td><strong>105.96s</strong></td>
</tr>
</tbody>
</table>

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9

- \( \times 7.63 \)
- \( \times 6.59 \)

### Characteristic Polynomial

- A new reduction to matrix multiplication in \( O(n^\omega) \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>1000</th>
<th>2000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>magma-v2.19-9</td>
<td>1.38s</td>
<td>24.28s</td>
<td>332.7s</td>
<td>2497s</td>
</tr>
<tr>
<td>fflas-ffpack</td>
<td><strong>0.532s</strong></td>
<td><strong>2.936s</strong></td>
<td><strong>32.71s</strong></td>
<td><strong>219.2s</strong></td>
</tr>
</tbody>
</table>

Intel Ivy-Bridge i5-3320 2.6Ghz using OpenBLAS-0.2.9

- \( \times 7.5 \)
- \( \times 6.7 \)
The case of Gaussian elimination

Which reduction to MatMul?

- Slab iterative
  - LAPACK
- Slab recursive
  - FFLAS-FFPACK
- Tile iterative
  - PLASMA
- Tile recursive
  - FFLAS-FFPACK
The case of Gaussian elimination

Which reduction to MatMul?

- Slab recursive
  FFLAS-FFPACK

- Tile recursive
  FFLAS-FFPACK

- Sub-cubic complexity: recursive algorithms
The case of Gaussian elimination

Which reduction to MatMul?

- Sub-cubic complexity: recursive algorithms
- Data locality
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

getrf: $A \rightarrow L, U$
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

\[
\text{trsm: } B \leftarrow BU^{-1}, B \leftarrow L^{-1}B
\]

\[
\text{gemm: } C \leftarrow C - A \times B
\]
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

\textbf{getrf}: \( A \rightarrow L, U \)

\textbf{trsm}: \( B \leftarrow BU^{-1}, B \leftarrow L^{-1}B \)

\textbf{gemm}: \( C \leftarrow C - A \times B \)
Block algorithms

Reductions and building blocks
In dense linear algebra

getrf: \( A \rightarrow L, U \)
trsm: \( B \leftarrow BU^{-1}, B \leftarrow L^{-1}B \)
gemm: \( C \leftarrow C - A \times B \)
Block algorithms

getrf: \( A \rightarrow L, U \)
Block algorithms

Tiled Iterative  Slab Recursive  Tiled Recursive

trsm: \( B \leftarrow BU^{-1}, B \leftarrow L^{-1}B \)
gemm: \( C \leftarrow C - A \times B \)
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

getrf: $A \rightarrow L, U$
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

trsm: \( B \leftarrow BU^{-1}, B \leftarrow L^{-1}B \)

gemm: \( C \leftarrow C - A \times B \)
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

getrf: $A \rightarrow L, U$
Block algorithms

Tiled Iterative

Trsm: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$

gemm: $C \leftarrow C - A \times B$
Block algorithms

getrf: $A \rightarrow L, U$
Block algorithms

getrf: $A \rightarrow L, U$
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

\[ \text{trsm: } B \leftarrow BU^{-1}, B \leftarrow L^{-1}B \]

\[ \text{gemm: } C \leftarrow C - A \times B \]
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

getrf: \( A \rightarrow L, U \)
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

\[ \text{trsm: } B \leftarrow BU^{-1}, B \leftarrow L^{-1}B \]

\[ \text{gemm: } C \leftarrow C - A \times B \]
**Block algorithms**

### Tiled Iterative

**getrf:** $A \to L, U$

**trsm:** $B \leftarrow BU^{-1}, B \leftarrow L^{-1}B$

**gemm:** $C \leftarrow C - A \times B$

### Slab Recursive

### Tiled Recursive
Block algorithms

getrf: $A \rightarrow L, U$
## Counting Modular Reductions

| $k \geq 1$ | Tiled Iter. Right looking | $\frac{1}{3k} n^3 + \left(1 - \frac{1}{k}\right) n^2 + \left(\frac{1}{6} k - \frac{5}{2} + \frac{3}{k}\right) n$ |
| $k \geq 1$ | Tiled Iter. Left looking | $(2 - \frac{1}{2k}) n^2 + \left(-\frac{5}{2} k - 1 + \frac{2}{k}\right) n + 2k^2 - 2k + 1$ |
| $k \geq 1$ | Tiled Iter. Crout | $(\frac{5}{2} - \frac{1}{k}) n^2 + \left(-2k - \frac{5}{2} + \frac{3}{k}\right) n + k^2$ |
### Counting Modular Reductions

<table>
<thead>
<tr>
<th>$k \geq 1$</th>
<th>Tiled Iter. Right looking</th>
<th>$\frac{1}{3k}n^3 + \left(1 - \frac{1}{k}\right)n^2 + \left(\frac{1}{6}k - \frac{5}{2} + \frac{3}{k}\right)n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k \geq 1$</td>
<td>Tiled Iter. Left looking</td>
<td>$(2 - \frac{1}{2k})n^2 + \left(-\frac{5}{2}k - 1 + \frac{2}{k}\right)n + 2k^2 - 2k + 1$</td>
</tr>
<tr>
<td>$k \geq 1$</td>
<td>Tiled Iter. Crout</td>
<td>$(\frac{5}{2} - \frac{1}{k})n^2 + \left(-2k - \frac{5}{2} + \frac{3}{k}\right)n + k^2$</td>
</tr>
<tr>
<td>$k = 1$</td>
<td>Iter. Right looking</td>
<td>$\frac{1}{3}n^3 - \frac{1}{3}n$</td>
</tr>
<tr>
<td>$k = 1$</td>
<td>Iter. Left Looking</td>
<td>$\frac{3}{2}n^2 - \frac{3}{2}n + 1$</td>
</tr>
<tr>
<td>$k = 1$</td>
<td>Iter. Crout</td>
<td>$\frac{3}{2}n^2 - \frac{7}{2}n + 3$</td>
</tr>
</tbody>
</table>
### Counting Modular Reductions

<table>
<thead>
<tr>
<th>$k \geq 1$</th>
<th>Tiled Iter. Right looking</th>
<th>$\frac{1}{3k} n^3 + (1 - \frac{1}{k}) n^2 + \left(\frac{1}{6} k - \frac{5}{2} + \frac{3}{k}\right) n$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tiled Iter. Left looking</td>
<td>$(2 - \frac{1}{2k}) n^2 + \left(-\frac{5}{2} k - 1 + \frac{2}{k}\right) n + 2k^2 - 2k + 1$</td>
</tr>
<tr>
<td></td>
<td>Tiled Iter. Crout</td>
<td>$(\frac{5}{2} - \frac{1}{k}) n^2 + \left(-2k - \frac{5}{2} + \frac{3}{k}\right) n + k^2$</td>
</tr>
<tr>
<td>$k = 1$</td>
<td>Iter. Right looking</td>
<td>$\frac{1}{3} n^3 - \frac{1}{3} n$</td>
</tr>
<tr>
<td></td>
<td>Iter. Left Looking</td>
<td>$\frac{3}{2} n^2 - \frac{3}{2} n + 1$</td>
</tr>
<tr>
<td></td>
<td>Iter. Crout</td>
<td>$\frac{3}{2} n^2 - \frac{7}{2} n + 3$</td>
</tr>
<tr>
<td></td>
<td>Tiled Recursive</td>
<td>$2n^2 - n \log_2 n - n$</td>
</tr>
<tr>
<td></td>
<td>Slab Recursive</td>
<td>$(1 + \frac{1}{4} \log_2 n) n^2 - \frac{1}{2} n \log_2 n - n$</td>
</tr>
</tbody>
</table>
Impact in practice

sequential LU decomposition variants on one core

- Right$(k=212)$
- Left$(k=212)$
- Crout$(k=212)$
- Tiled-Rec
- Slab-Rec

As anticipated: Right-looking $<$ Crout $<$ Left-looking
Impact in practice

As anticipated: Right-looking < Crout < Left-looking

C. Pernet
Exact Linear Algebra Algorithmic
July 6, 2015 37 / 73
Dealing with rank deficiencies and computing rank profiles

Rank profiles: first linearly independent columns

- Major invariant of a matrix (echelon form)
- Gröbner basis computations (Macaulay matrix)
- Krylov methods

Gaussian elimination revealing echelon forms:

[Ibarra, Moran and Hui 82]

[Keller-Gehrig 85]

[Jeannerod, P. and Storjohann 13]
Computing rank profiles

Lessons learned (or what we thought was necessary):

- treat rows in order
- exhaust all columns before considering the next row
- **slab** block splitting required (recursive or iterative)
  ↜ similar to partial pivoting
Computing rank profiles

Lessons learned (or what we thought was necessary):

- treat rows in order
- exhaust all columns before considering the next row
- slab block splitting required (recursive or iterative)
  \[\leadsto\] similar to partial pivoting

Tiled recursive PLUQ [Dumas P. Sultan 13,15]

1. Generalized to handle rank deficiency
   - 4 recursive calls necessary
   - in-place computation

2. Pivoting strategies exist to recover rank profile and echelon forms
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

2 × 2 block splitting
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

Recursive call
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

\[ B \leftarrow BU^{-1} \]
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

\[
\begin{align*}
\text{TRSM: } & B \leftarrow L^{-1}B \\
\end{align*}
\]
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

MatMul: $C \leftarrow C - A \times B$
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

MatMul: $C \leftarrow C - A \times B$
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

\[
\text{MatMul: } C \leftarrow C - A \times B
\]
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

2 independent recursive calls

\[ O(mn\omega - 2) \] (degenerating to \( 2/3n^3 \))

computing col. and row rank profiles of all leading sub-matrices

fewer modular reductions than slab algorithms

rank deficiency introduces parallelism
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

TRSM: $B \leftarrow BU^{-1}$
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

\[
\text{TRSM: } B \leftarrow L^{-1}B
\]
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

MatMul: $C \leftarrow C - A \times B$
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

\[ \text{MatMul: } C \leftarrow C - A \times B \]
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

**MatMul**: $C \leftarrow C - A \times B$
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

Recursive call
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

Puzzle game (block cyclic rotations)
A tiled recursive algorithm

[Dumas, P. and Sultan 13]

- $O(mnr^{\omega-2})$ (degenerating to $2/3n^3$)
- computing col. and row rank profiles of all leading sub-matrices
- fewer modular reductions than slab algorithms
- rank deficiency introduces parallelism
Computing all rank profiles at once

Dumas, P. and Sultan ISSAC’15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $R_A \in \{0, 1\}^{m \times n}$ such that any pair of $(i, j)$-leading sub-matrix of $R_A$ and of $A$ have the same rank.

\[
\begin{pmatrix}
1 & 2 & 3 & 4 \\
2 & 4 & 5 & 8 \\
1 & 2 & 3 & 4 \\
3 & 5 & 9 & 12
\end{pmatrix}
\quad \rightarrow \quad
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix}
\]
Computing all rank profiles at once

Dumas, P. and Sultan ISSAC’15 (Thursday 9 @ 3PM )

Definition (Rank Profile matrix)

The unique $R_A \in \{0, 1\}^{m \times n}$ such that any pair of $(i, j)$-leading sub-matrix of $R_A$ and of $A$ have the same rank.

Theorem

- RowRP and ColRP read directly on $R(A)$
- Same holds for any $(i, j)$-leading submatrix.

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>2 4 5 8</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>1 2 3 4</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>3 5 9 12</td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>

RowRP = \{1\}  
ColRP = \{1\}
Computing all rank profiles at once

Dumas, P. and Sultan ISSAC’15 (Thursday 9 @ 3PM )

Definition (Rank Profile matrix)

The unique $R_A \in \{0, 1\}^{m \times n}$ such that any pair of $(i, j)$-leading sub-matrix of $R_A$ and of $A$ have the same rank.

Theorem

- RowRP and ColRP read directly on $R(A)$
- Same holds for any $(i, j)$-leading submatrix.

RowRP = \{1,2\}
ColRP = \{1,3\}
Computing all rank profiles at once

Dumas, P. and Sultan ISSAC’15 (Thursday 9 @ 3PM )

Definition (Rank Profile matrix)

The unique $\mathcal{R}_A \in \{0, 1\}^{m \times n}$ such that any pair of $(i, j)$-leading sub-matrix of $\mathcal{R}_A$ and of $A$ have the same rank.

Theorem

- RowRP and ColRP read directly on $\mathcal{R}(A)$
- Same holds for any $(i, j)$-leading submatrix.

RowRP = \{1,4\}
ColRP = \{1,2\}
Computing all rank profiles at once

Dumas, P. and Sultan ISSAC’15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $R_A \in \{0, 1\}^{m \times n}$ such that any pair of $(i,j)$-leading sub-matrix of $R_A$ and of $A$ have the same rank.

Theorem

- RowRP and ColRP read directly on $R(A)$
- Same holds for any $(i,j)$-leading submatrix.

$$A = PLUQ = P \begin{bmatrix} L & 0 \\ M & I_{m-r} \end{bmatrix} \begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} U & V \\ I_{n-r} & 0 \end{bmatrix} Q$$
Computing all rank profiles at once

Dumas, P. and Sultan ISSAC’15 (Thursday 9 @ 3PM )

Definition (Rank Profile matrix)

The unique $R_A \in \{0, 1\}^{m \times n}$ such that any pair of $(i, j)$-leading sub-matrix of $R_A$ and of $A$ have the same rank.

Theorem

- RowRP and ColRP read directly on $R(A)$
- Same holds for any $(i, j)$-leading submatrix.

$$A = PLUQ = P \begin{bmatrix} L & 0 \\ M & I_{m-r} \end{bmatrix} P^T P \begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix} Q Q^T \begin{bmatrix} U & V \\ I_{n-r} \end{bmatrix} Q$$

RowRP = $\{1, 4\}$
ColRP = $\{1, 2\}$
Computing all rank profiles at once

Dumas, P. and Sultan ISSAC’15 (Thursday 9 @ 3PM )

Definition (Rank Profile matrix)

The unique $R_A \in \{0, 1\}^{m \times n}$ such that any pair of $(i, j)$-leading sub-matrix of $R_A$ and of $A$ have the same rank.

Theorem

- RowRP and ColRP read directly on $R(A)$
- Same holds for any $(i, j)$-leading submatrix.

$$A = PLUQ = P \begin{bmatrix} L & 0 \\ M & I_{m-r} \end{bmatrix} P^T P \begin{bmatrix} I_r & 0 \\ \Pi_{P,Q} \end{bmatrix} Q Q^T \begin{bmatrix} U & V \\ I_{n-r} & \end{bmatrix} Q$$
Computing all rank profiles at once

Dumas, P. and Sultan ISSAC’15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $R_A \in \{0, 1\}^{m \times n}$ such that any pair of $(i, j)$-leading sub-matrix of $R_A$ and of $A$ have the same rank.

Theorem

- RowRP and ColRP read directly on $R(A)$
- Same holds for any $(i, j)$-leading submatrix.

$$A = PLUQ = P \begin{bmatrix} L & 0 \\ M & I_{m-r} \end{bmatrix} P^T P \begin{bmatrix} I_r & 0 \\ & 0 \end{bmatrix} Q Q^T \begin{bmatrix} U & V \\ & I_{n-r} \end{bmatrix} Q$$

With appropriate pivoting: $\Pi_{P,Q} = R(A)$
Reductions in black box linear algebra

Matrix-Vector Product: building block,
\[ \sim \text{costs } E(n) \]

Minimal polynomial: [Wiedemann 86]
\[ \sim \text{iterative Krylov/Lanczos methods} \]
\[ \sim O(nE(n) + n^2) \]
Matrix-Vector Product: building block, 
\( \sim \) costs \( E(n) \)

Minimal polynomial: [Wiedemann 86]
\( \sim \) iterative Krylov/Lanczos methods
\( \sim O(nE(n) + n^2) \)

Rank, Det, Solve: [Chen & Al. 02]
\( \sim \) reduces to MinPoly + preconditioners
\( \sim O^*(nE(n) + n^2) \)
Reductions and building blocks

Reductions in black box linear algebra

Matrix-Vector Product: building block,
\[ \sim \] costs \( E(n) \)

Minimal polynomial: [Wiedemann 86]
\[ \sim \] iterative Krylov/Lanczos methods
\[ \sim O(nE(n) + n^2) \]

Rank, Det, Solve: [Chen & Al. 02]
\[ \sim \] reduces to MinPoly + preconditioners
\[ \sim O(nE(n) + n^2) \]

Characteristic Poly.: [Dumas P. Saunders 09]
\[ \sim \] reduces to MinPoly, Rank, \ldots
Reductions and building blocks

Reductions in black box linear algebra

Matrix-Vector Product: building block, \( \leadsto \) costs \( E(n) \)

Minimal polynomial: [Wiedemann 86]
\( \leadsto \) iterative Krylov/Lanczos methods
\( \leadsto O(nE(n) + n^2) \)

Rank, Det, Solve: [Chen & Al. 02]
\( \leadsto \) reduces to MinPoly + preconditioners
\( \leadsto O^\sim(nE(n) + n^2) \)

Characteristic Poly.: [Dumas P. Saunders 09]
\( \leadsto \) reduces to MinPoly, Rank, …
Outline

1. Choosing the underlying arithmetic
   - Using boolean arithmetic
   - Using machine word arithmetic
   - Larger field sizes

2. Reductions and building blocks
   - In dense linear algebra
   - In blackbox linear algebra

3. Size dimension trade-offs
   - Hermite normal form
   - Frobenius normal form

4. Parallel exact linear algebra
   - Ingredients for the parallelization
   - Parallel dense linear algebra mod $p$
Size Dimension trade-offs

Computing with coefficients of varying size: \( \mathbb{Z}, \mathbb{Q}, K[X], \ldots \)

Multimodular methods

- over \( K[X] \): evaluation-interpolation
- over \( \mathbb{Z}, \mathbb{Q} \): Chinese Remainder Theorem

\[
\text{Cost} = \text{Algebraic Cost} \times \text{Size(Output)}
\]

✓ avoids coefficient blow-up

✗ uniform (worst case) cost for all arithmetic ops
Size Dimension trade-offs

Computing with coefficients of varying size: \( \mathbb{Z}, \mathbb{Q}, K[X], \ldots \)

**Multimodular methods**
- over \( K[X] \): evaluation-interpolation
- over \( \mathbb{Z}, \mathbb{Q} \): Chinese Remainder Theorem

\[
\text{Cost} = \text{Algebraic Cost} \times \text{Size(Output)}
\]

- \(\checkmark\) avoids coefficient blow-up
- \(\times\) uniform (worst case) cost for all arithmetic ops

**Example**

Hadamard’s bound: \(|\det(A)| \leq (\|A\|_\infty \sqrt{n})^n.\)

\(\text{LinSys}_{\mathbb{Z}}(n) = O(n^\omega \times n(\log n + \log \|A\|_\infty))\)
Size Dimension trade-offs

Computing with coefficients of varying size: \( \mathbb{Z}, \mathbb{Q}, K[X], \ldots \)

**Multimodular methods**

- over \( K[X] \): evaluation-interpolation
- over \( \mathbb{Z}, \mathbb{Q} \): Chinese Remainder Theorem

\[
\text{Cost} = \text{Algebraic Cost} \times \text{Size(Output)}
\]

✓ avoids coefficient blow-up

✗ uniform (worst case) cost for all arithmetic ops

**Example**

Hadamard’s bound: \( |\det(A)| \leq (\|A\|_\infty \sqrt{n})^n \).

LinSys\(_{\mathbb{Z}}(n) = O(n^\omega \times n(\log n + \log \|A\|_\infty)) = O^\sim(n^{\omega+1} \log \|A\|_\infty)\]
Size Dimension trade-offs

Computing with coefficients of varying size: $\mathbb{Z}, \mathbb{Q}, K[X], \ldots$

**Lifting techniques**

$p$-adic lifting: [Moenck & Carter 79, Dixon 82]
- One computation over $\mathbb{Z}_p$
- Iterative lifting of the solution to $\mathbb{Z}, \mathbb{Q}$

**Example**

$$\text{LinSys}_{\mathbb{Z}}(n) = O(n^3 \log \|A\|^{1+\epsilon})$$
Size Dimension trade-offs

Computing with coefficients of varying size: $\mathbb{Z}, \mathbb{Q}, K[X], \ldots$

Lifting techniques

$p$-adic lifting: [Moenck & Carter 79, Dixon 82]
- One computation over $\mathbb{Z}_p$
- Iterative lifting of the solution to $\mathbb{Z}, \mathbb{Q}$

High order lifting: [Storjohann 02,03]
- Fewer iteration steps
- Larger dimension in the lifting

Example

$\text{LinSys}_{\mathbb{Z}}(n) = \mathcal{O}(n^\omega \log \|A\|_\infty)$
Improving time Complexities

Matrix multiplication: door to fast linear algebra

- over $\mathbb{Z}$: $O(n^\omega M(\log \|A\|)) = O^*(n^\omega \log \|A\|)$
Improving time Complexities

Matrix multiplication: door to fast linear algebra

- over \( \mathbb{Z} \): \( O(n^\omega M (\log \|A\|)) = O^\sim(n^\omega \log \|A\|) \)

Equivalence over \( \mathbb{Z} \) or \( K[X] \): Hermite normal form

- \([\text{Kannan & Bachem 79}]\):
- \([\text{Chou & Collins 82}]\):
- \([\text{Domich & Al. 87}, \text{Illicopoulos 89}]\):
- \([\text{Micciancio & Warinschi 01}]\):
- \([\text{Storjohann & Labahn 96}]\):
- \([\text{Gupta & Storjohann 11}]\):

\( O^\sim(n^6 \log \|A\|) \) ∈ \( P \)
\( O^\sim(n^4 \log \|A\|) \)
\( O^\sim(n^5 \log \|A\|^2) \)
\( O^\sim(n^3 \log \|A\|) \) heur.
\( O^\sim(n^\omega + 1 \log \|A\|) \)
\( O^\sim(n^3 \log \|A\|) \)
Improving time Complexities

Matrix multiplication: door to fast linear algebra
- over \( \mathbb{Z} \): \( O(n^\omega M(\log \|A\|)) = O^\sim(n^\omega \log \|A\|) \)

Equivalence over \( \mathbb{Z} \) or \( K[X] \): Hermite normal form
- [Kannan & Bachem 79]:
- [Chou & Collins 82]:
- [Domich & Al. 87], [Illiopoulos 89]:
- [Micciancio & Warinschi 01]:
  - \( O^\sim(n^6 \log \|A\|) \)
  - \( O^\sim(n^4 \log \|A\|) \)
  - \( O^\sim(n^5 \log \|A\|^2), O^\sim(n^3 \log \|A\|) \) heur.
- [Storjohann & Labahn 96]:
  - \( O^\sim(n^{\omega+1} \log \|A\|) \)
- [Gupta & Storjohann 11]:

Similarity over a field: Frobenius normal form
- [Giesbrecht 93]:
- [Storjohann 00]:
- [P. & Storjohann 07]:
  - \( O^\sim(n^\omega) \) probabilistic
  - \( O^\sim(n^\omega) \) deterministic
  - \( O(n^\omega) \) probabilistic
Improving time Complexities

Matrix multiplication: door to fast linear algebra

- over $\mathbb{Z}$: $O(n^\omega M (\log \| A \|)) = O^\sim(n^\omega \log \| A \|)$

Equivalence over $\mathbb{Z}$ or $K[X]$: Hermite normal form

- [Kannan & Bachem 79]: $\in P$
- [Chou & Collins 82]: $O^\sim(n^6 \log \| A \|)$
- [Domich & Al. 87], [Illiopoulos 89]: $O^\sim(n^4 \log \| A \|)$
- [Micciancio & Warinschi 01]: $O^\sim(n^5 \log \| A \|^2)$, $O^\sim(n^3 \log \| A \|)$ heur.
- [Storjohann & Labahn 96]: $O^\sim(n^{\omega+1} \log \| A \|)$
- [Gupta & Storjohann 11]: $O^\sim(n^3 \log \| A \|)$

Similarity over a field: Frobenius normal form

- [Giesbrecht 93]: $O^\sim(n^\omega)$ probabilistic
- [Storjohann 00]: $O^\sim(n^\omega)$ deterministic
- [P. & Storjohann 07]: $O(n^\omega)$ probabilistic
Building blocks and reductions

In brief

Reductions to a building block

Matrix Mult: block rec. \( \sum_{i=1}^{\log n} n \left( \frac{n}{2^i} \right)^{\omega-1} = O(n^\omega) \) (Gauss, REF)

Matrix Mult: Iterative \( \sum_{k=1}^{n} k \left( \frac{n}{k} \right)^{\omega} = O(n^\omega) \) (Frobenius)

Linear Sys: over \( \mathbb{Z} \) (Hermite Normal Form)

Size/dimension compromises

- Hermite normal form: rank 1 updates reducing the determinant
- Frobenius normal form: degree \( k \), dimension \( n/k \) for \( k = 1 \ldots n \)
Hermite normal form: naive algorithm

Reduced Echelon form over a ring:

\[
\begin{bmatrix}
p_1 & * & x_{1,2} & * & * & x_{1,3} & * \\
p_2 & * & * & x_{2,3} & * \\
p_3 & * & & & & &
\end{bmatrix}
\]

with

\[0 \leq x_{*,j} < p_j.\]

for \(i = 1 \ldots n\) do

\[(g, t_i, \ldots, t_n) = \gcd(A_{i,i}, A_{i+1,i}, \ldots, A_{n,i})\]

\(L_i \leftarrow \sum_{j=i+1}^{n} t_j L_j\)

for \(j = i + 1 \ldots n\) do

\(L_j \leftarrow L_j - \frac{A_{j,i}}{g} L_i\)

end for

for \(j = 1 \ldots i - 1\) do

\(L_j \leftarrow L_j - \lfloor \frac{A_{j,i}}{g} \rfloor L_i\)

end for

end for

▷ eliminate

▷ reduce
Computing modulo the determinant [Domich & Al. 87]

Property

For a non-singular: \( \max_i \sum_j H_{ij} \leq \det H = \det A \)

Example

\[
A = \begin{bmatrix}
-5 & 8 & -3 & -9 & 5 & 5 \\
-2 & 8 & -2 & -2 & 8 & 5 \\
7 & -5 & -8 & 4 & 3 & -4 \\
1 & -1 & 6 & 0 & 8 & -3
\end{bmatrix},
H = \begin{bmatrix}
1 & 0 & 3 & 237 & -299 & 90 \\
0 & 1 & 1 & 103 & -130 & 40 \\
0 & 0 & 4 & 352 & -450 & 135 \\
0 & 0 & 0 & 486 & -627 & 188
\end{bmatrix}
\]

\[\det A = 1944\]
Computing modulo the determinant [Domich & Al. 87]

Property

- For $A$ non-singular: $\max_i \sum_j H_{ij} \leq \det H = \det A$
- Every computation can be done modulo $d = \det A$:

\[
U' \begin{bmatrix} A \\ dI_n \\ I_n \end{bmatrix} = \begin{bmatrix} H \\ I_n \end{bmatrix}
\]

Example

\[
A = \begin{bmatrix}
-5 & 8 & -3 & -9 & 5 & 5 \\
-2 & 8 & -2 & -2 & 8 & 5 \\
7 & -5 & -8 & 4 & 3 & -4 \\
1 & -1 & 6 & 0 & 8 & -3 
\end{bmatrix},
\quad
H = \begin{bmatrix}
1 & 0 & 3 & 237 & -299 & 90 \\
0 & 1 & 1 & 103 & -130 & 40 \\
0 & 0 & 4 & 352 & -450 & 135 \\
0 & 0 & 0 & 486 & -627 & 188 
\end{bmatrix}
\]

\[
\det A = 1944
\]

\[
\leadsto O(n^3) \times M(n(\log n + \log \|A\|)) = O^\sim(n^5 \log \|A\|^2)
\]
Computing modulo the determinant

- Pessimistic estimate on the arithmetic size
- $d$ large but most coefficients of $H$ are small
- On average: only the last few columns are large

Compute $H'$ close to $H$ but with small determinant
Computing modulo the determinant

- Pessimistic estimate on the arithmetic size
- \( d \) large but most coefficients of \( H \) are small
- On average: only the last few columns are large

\[ A = \begin{bmatrix} B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix} \]

\( d_1 = \det \left( \begin{bmatrix} B \\ c^T \end{bmatrix} \right) \), \( d_2 = \det \left( \begin{bmatrix} B \\ d^T \end{bmatrix} \right) \)

\( g = \gcd(d_1, d_2) = s d_1 + t d_2 \) Then

\[ \det \left( \begin{bmatrix} B & c^T \\ s c^T + td^T \end{bmatrix} \right) = g \]
Micciancio & Warinschi algorithm

Compute $d_1 = \det \left( \begin{bmatrix} B \\ c^T \end{bmatrix} \right), d_2 = \det \left( \begin{bmatrix} B \\ d^T \end{bmatrix} \right)$ $\triangleright$ Double Det

$(g, s, t) = \gcd(d_1, d_2)$

Compute $H_1$ the HNF of $\begin{bmatrix} B \\ sc^T + td^T \end{bmatrix} \mod g$ $\triangleright$ Modular HNF

Recover $H_2$ the HNF of $\begin{bmatrix} B \\ sc^T + td^T \end{bmatrix} \begin{bmatrix} b \\ sa_{n-1,n} + ta_{n,n} \end{bmatrix}$ $\triangleright$ AddCol

Recover $H_3$ the HNF of $\begin{bmatrix} B \\ c^T \\ d^T \end{bmatrix} \begin{bmatrix} b \\ a_{n-1,n} \\ a_{n,n} \end{bmatrix}$ $\triangleright$ AddRow
Micciancio & Warinschi algorithm

Compute $d_1 = \det\left(\begin{bmatrix} B \\ c^T \end{bmatrix}\right)$, $d_2 = \det\left(\begin{bmatrix} B \\ d^T \end{bmatrix}\right)$

$(g, s, t) = \text{xgcd}(d_1, d_2)$

Compute $H_1$ the HNF of $\begin{bmatrix} B \\ sc^T + td^T \end{bmatrix}$ mod $g$

Recover $H_2$ the HNF of $\begin{bmatrix} B \\ sc^T + td^T & sa_{n-1,n} + ta_{n,n} \\ B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix}$

Recover $H_3$ the HNF of $\begin{bmatrix} B \\ sc^T + td^T \\ B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix}^\Delta$

\[ \Rightarrow \text{Double Det} \]
\[ \Rightarrow \text{Modular HNF} \]
\[ \Rightarrow \text{AddCol} \]
\[ \Rightarrow \text{AddRow} \]
Double Determinant

First approach: LU mod $p_1, \ldots, p_k + \text{CRT}$

- Only one elimination for the $n-2$ first rows
- 2 updates for the last rows (triangular back substitution)
- $k$ large such that $\prod_{i=1}^{k} p_i > n^n \log \|A\|^{n/2}$

Second approach: [Abbott Bronstein Mulders 99]

- Solve $Ax = b$.
- $\delta = \text{lcm}(q_1, \ldots, q_n)$ s.t. $x_i = p_i / q_i$
- Then $\delta$ is a large divisor of $D = \det A$.

- Compute $D/\delta$ by LU mod $p_1, \ldots, p_k + \text{CRT}$
- $k$ small, such that $\prod_{i=1}^{k} p_i > n^n \log \|A\|^{n/2}$
Double Determinant

First approach: LU mod $p_1, \ldots, p_k +$ CRT

- Only one elimination for the $n - 2$ first rows
- 2 updates for the last rows (triangular back substitution)
- $k$ large such that $\prod_{i=1}^{k} p_i > n^n \log \|A\|^{n/2}$

Second approach: [Abbott Bronstein Mulders 99]

- Solve $Ax = b$.
- $\delta = \text{lcm}(q_1, \ldots, q_n)$ s.t. $x_i = p_i/q_i$

Then $\delta$ is a large divisor of $D = \det A$.

- Compute $D/\delta$ by LU mod $p_1, \ldots, p_k +$ CRT
- $k$ small, such that $\prod_{i=1}^{k} p_i > n^n \log \|A\|^{n/2}/\delta$
Double Determinant: improved

Property

Let \( x = [x_1, \ldots, x_n] \) be the solution of \( [A \mid c] x = d \). Then \( y = [-\frac{x_1}{x_n}, \ldots, -\frac{x_{n-1}}{x_n}, \frac{1}{x_n}] \) is the solution of \( [A \mid d] y = c \).

- 1 system solve
- 1 LU for each \( p_i \)

\( d_1, d_2 \) computed at about the cost of 1 déterminant
**AddCol**

**Problem**

Find a vector $e$ such that

$$
\begin{bmatrix}
H_1 & e
\end{bmatrix} = U \begin{bmatrix}
B_{sc^T + td^T} & b \\
\end{bmatrix}
$$

$$
e = U \begin{bmatrix}
b \\
\end{bmatrix}
= \begin{bmatrix}
b \\
\end{bmatrix}
= H_1 \begin{bmatrix}
B_{sc^T + td^T}^{-1} & b \\
\end{bmatrix}
\begin{bmatrix}
sa_{n-1,n} + ta_{n,n}
\end{bmatrix}
$$

⇝ Solve a system.

- $n - 1$ first rows are small
- last row is large
AddCol

Idea:
replace the last row by a random small one $w^T$.

$$\begin{bmatrix} B \\ w^T \end{bmatrix} y = \begin{bmatrix} b \\ a_{n-1,n-1} \end{bmatrix}$$

Let $\{k\}$ be a basis of the kernel of $B$. Then

$$x = y + \alpha k.$$ 

where

$$\alpha = \frac{a_{n-1,n-1} - (sc^T + td^T) \cdot y}{(sc^T + td^T) \cdot k}$$

⇝ limits the expensive arithmetic to a few dot products
Computing the Frobenius normal form

**Definition**

Unique $F = U^{-1}AU = \text{Diag}(C_{f_0}, \ldots, C_{f_k})$ with $f_k | f_{k-1} | \cdots | f_0$. 
Computing the Frobenius normal form

[P. & Storjohann 07]

$k$-shifted form:

\[
\begin{array}{ccc}
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{ccc}
k & k & \leq k \\
k & k & \leq k \\
\end{array}
\]

From $k$ to $k + 1$-shifted in $O(k(n^k)\omega)$

Compute iteratively from a $1$-shifted form

Invariant factors appear by increasing degree

Until the Hessenberg polycyclic form

$\sum_{k=1}^{n\omega}(\omega - 1)\leq\zeta(\omega - 1)\omega = O(n\omega)$

Generalized to the Frobenius form as well

Transformation matrix in $O(n\omega \log \log n)$
Computing the Frobenius normal form

[P. & Storjohann 07]

$k + 1$-shifted form:
Computing the Frobenius normal form

[P. & Storjohann 07]

\[
\begin{array}{ccc}
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0
\end{array}
\]

- From \( k \) to \( k + 1 \)-shifted in \( O(k(n/k)^\omega) \)
- Compute iteratively from a 1-shifted form
- Invariant factors appear by increasing degree
Computing the Frobenius normal form

[P. & Storjohann 07]

<table>
<thead>
<tr>
<th>Hessenberg polycyclic:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 0</td>
</tr>
<tr>
<td>1 1 1</td>
</tr>
</tbody>
</table>

- From $k$ to $k + 1$-shifted in $O(k \left( \frac{n}{k} \right)^\omega)$
- Compute iteratively from a 1-shifted form
- Invariant factors appear by increasing degree
- Until the Hessenberg polycyclic form
Computing the Frobenius normal form

[Hessenberg polycyclic:]

- From $k$ to $k + 1$-shifted in $O(k (\frac{n}{k})^\omega)$
- Compute iteratively from a 1-shifted form
- Invariant factors appear by increasing degree
- Until the Hessenberg polycyclic form

$$n^\omega \sum_{k=1}^{n} \left( \frac{1}{k} \right)^{\omega-1} \leq \zeta (\omega - 1)n^\omega = O(n^\omega)$$
Computing the Frobenius normal form

[P. & Storjohann 07]

Hessenberg polycyclic:

- From $k$ to $k + 1$-shifted in $O(k (\frac{n}{k})^\omega)$
- Compute iteratively from a 1-shifted form
- Invariant factors appear by increasing degree
- Until the Hessenberg polycyclic form

$$n^\omega \sum_{k=1}^{n} \left( \frac{1}{k} \right)^{\omega - 1} \leq \zeta (\omega - 1) n^\omega = O(n^\omega)$$

- Generalized to the Frobenius form as well
- Transformation matrix in $O(n^\omega \log \log n)$
A new type size dimension trade-off

\[ xI_n - A \]

- dimension = \( n \)
- degree = 1

\[ \text{det}(xI_n - A) \]

- dimension = 1
- degree = \( n \)
A new type size dimension trade-off

\[ xI_n - A \]

\[ \det(xI_n - A) \]

**Keller-Gehrig 2**

- Dimension = \( n \)
- Degree = 1

- Dimension = \( \frac{n}{2^i} \)
- Degree = \( 2^i \)

- Dimension = 1
- Degree = \( n \)
A new type size dimension trade-off

\[ xI_n - A \]
\[ \text{dimension} = n \]
\[ \text{degree} = 1 \]

\[ \det(xI_n - A) \]
\[ \text{dimension} = 1 \]
\[ \text{degree} = n \]

Keller-Gehrig 2

\[ \frac{n}{2^i} \]
\[ \text{degree} = 2^i \]

New algorithm

\[ \frac{n}{k} \]
\[ \text{degree} = k \]
Outline

1. Choosing the underlying arithmetic
   - Using boolean arithmetic
   - Using machine word arithmetic
   - Larger field sizes

2. Reductions and building blocks
   - In dense linear algebra
   - In blackbox linear algebra

3. Size dimension trade-offs
   - Hermite normal form
   - Frobenius normal form

4. Parallel exact linear algebra
   - Ingredients for the parallelization
   - Parallel dense linear algebra mod $p$
Parallel numerical linear algebra

- cost invariant wrt. splitting
  - $O(n^3)$
  - $\leadsto$ fine grain
  - $\leadsto$ block iterative algorithms
- regular task load
- Numerical stability constraints
Parallel exact linear algebra

Ingredients for the parallelization

Parallelization

Parallel numerical linear algebra

- cost invariant wrt. splitting
  - $O(n^3)$
  - $\leadsto$ fine grain
  - $\leadsto$ block iterative algorithms
- regular task load
- Numerical stability constraints

Exact linear algebra specificities

- cost affected by the splitting
  - $O(n^\omega)$ for $\omega < 3$
  - modular reductions
  - $\leadsto$ coarse grain
  - $\leadsto$ recursive algorithms
- rank deficiencies
  - $\leadsto$ unbalanced task loads
Ingredients for the parallelization

Criteria

- good performances
- portability across architectures
- abstraction for simplicity

Challenging key point: scheduling as a plugin

Program: only describes where the parallelism lies

Runtime: scheduling & mapping, depending on the context of execution

3 main models:

1. Parallel loop [data parallelism]
2. Fork-Join (independent tasks) [task parallelism]
3. Dependent tasks with data flow dependencies [task parallelism]
Data Parallelism

**OMP**

```c
for (int step = 0; step < 2; ++step){
    #pragma omp parallel for
    for (int i = 0; i < count; ++i)
        A[i] = (B[i+1] + B[i-1] + 2.0*B[i])*0.25;
}
```

**Limitation:** very un-efficient with recursive parallel regions

- Limited to iterative algorithms
- No composition of routines
Task parallelism with fork-Join

- Task based program: `spawn + sync`
- Especially suited for recursive programs

**OMP (since v3)**

```c
void fibonacci (long* result, long n) {
    if (n < 2)
        *result = n;
    else {
        long x, y;
        #pragma omp task
        fibonacci (&x, n-1);
        fibonacci (&y, n-2);
        #pragma omp taskwait
        *result = x + y;
    }
}
```
Tasks with dataflow dependencies

➤ Task based model avoiding synchronizations
➤ Infer synchronizations from the read/write specifications
  ▶ A task is ready for execution when all its inputs variables are ready
  ▶ A variable is ready when it has been written
➤ Recently supported: Athapascan [96], Kaapi [06], StarSs [07], StarPU [08], Quark [10], OMP since v4 [14]...
Illustration: Cholesky factorization

```c
void Cholesky( double* A, int N, size_t NB ) { 

    for (size_t k=0; k < N; k += NB) 
    {
        clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

        for (size_t m=k+ NB; m < N; m += NB) 
        {
            cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit, 
                          NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
        }

        for (size_t m=k+ NB; m < N; m += NB) 
        {
            cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans, 
                           NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+N+m], N );

            for (size_t n=k+NB; n < m; n += NB) 
            {
                cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans, 
                              NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
            }
        }
    }
}
```
Illustration: Cholesky factorization

```c
void Cholesky( double* A, int N, size_t NB ) {
    #pragma omp parallel
    #pragma omp single nowait
    for (size_t k=0; k < N; k += NB) {
        clapack_dpotrf(CblasRowMajor, CblasLower, NB, &A[k*N+k], N);
        for (size_t m=k+ NB; m < N; m += NB) {
            #pragma omp task firstprivate(k, m) shared(A)
            cblas_dtrsm(CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
                        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N);
        }
        #pragma omp taskwait // Barrier: no concurrency with next tasks
        for (size_t m=k+ NB; m < N; m += NB) {
            #pragma omp task firstprivate(k, m) shared(A)
            cblas_dsyrk(CblasRowMajor, CblasLower, CblasNoTrans, CblasTrans,
                        NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N);
            for (size_t n=k+NB; n < m; n += NB) {
                #pragma omp task firstprivate(k, m) shared(A)
                cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans,
                            NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N);
            }
        }
        #pragma omp taskwait // Barrier: no concurrency with tasks at iteration k+1
    }
}
```
Parallel exact linear algebra

Ingredients for the parallelization
Parallel exact linear algebra

Ingredients for the parallelization

C. Pernet

Exact Linear Algebra Algorithmic

July 6, 2015

66 / 73
SYNC.
Illustration: Cholesky factorization

```c
void Cholesky( double* A, int N, size_t NB ){
    #pragma kaapi parallel
    for ( size_t k=0; k < N; k += NB )
    {
        #pragma kaapi task readwrite(&A[k*N+k]{ld=N; [NB][NB]})
        clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

        for ( size_t m=k+ NB; m < N; m += NB )
        {
            #pragma kaapi task read(&A[k*N+k]{ld=N; [NB][NB]}) readwrite(&A[m*N+k]{ld=N; [NB][NB]})
            cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit, 
            NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
        }

        for ( size_t m=k+ NB; m < N; m += NB )
        {
            #pragma kaapi task read(&A[m*N+k]{ld=N; [NB][NB]}) readwrite(&A[m*N+m]{ld=N; [NB][NB]})
            cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans, 
            NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

        for ( size_t n=k+NB; n < m; n += NB )
        {
            #pragma kaapi task read(&A[m*N+k]{ld=N; [NB][NB]}, &A[n*N+k]{ld=N; [NB][NB]})
            readwrite(&A[m*N+n]{ld=N; [NB][NB]})
            cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans, 
            NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
        }
    }

    // Implicit barrier only at the end of Kaapi parallel region
}
```
Parallel matrix multiplication

[Dumas, Gautier, P. & Sultan 14]
Parallel matrix multiplication

[Dumas, Gautier, P. & Sultan 14]

\[
\begin{array}{c|c}
A_1 & C_{11} \quad C_{12} \\
\hline
A_2 & C_{21} \quad C_{22}
\end{array}
\]

1st recursion cutting
2nd recursion cutting

Graph showing performance comparison of pfgemm over \( \mathbb{Z}/131071\mathbb{Z} \) on a Xeon E5-4620 2.2Ghz 32 cores with various libraries.
Parallel matrix multiplication

[Dumas, Gautier, P. & Sultan 14]
Gaussian elimination

- Slab iterative
  - LAPACK
- Slab recursive
  - FFLAS-FFPACK
- Tile iterative
  - PLASMA
- Tile recursive
  - FFLAS-FFPACK
Gaussian elimination

- Prefer recursive algorithms
Gaussian elimination

- Prefer recursive algorithms
- Better data locality
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Comparing numerical efficiency (no modulo)
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Comparing numerical efficiency (no modulo)
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Comparing numerical efficiency (no modulo)
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Over the finite field $\mathbb{Z}/131071\mathbb{Z}$
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Over the finite field $\mathbb{Z}/131071\mathbb{Z}$
Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- So far, **floating point** arithmetic delivers best speed
Conclusion

**Design framework for high performance exact linear algebra**

Asymptotic reduction $\succ$ algorithm tuning $\succ$ building block implementation

- So far, **floating point** arithmetic delivers best speed
- Medium size arithmetic: **RNS**
  - harnesses floating point efficiency
  - embarrassingly easy parallelization (and fault tolerance)
Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction $> \text{algorithm tuning} > \text{building block implementation}$

- So far, **floating point** arithmetic delivers best speed
- Medium size arithmetic: **RNS**
  $\leadsto$ harnesses floating point efficiency
  $\leadsto$ embarrassingly easy parallelization (and fault tolerance)
- Favor **tiled recursive** algorithms
  $\leadsto$ **architecture oblivious vs aware** algorithms [Gustavson 07]
Design framework for high performance exact linear algebra

Asymptotic reduction $\gg$ algorithm tuning $\gg$ building block implementation

- So far, **floating point** arithmetic delivers best speed
- Medium size arithmetic: **RNS**
  - $\Rightarrow$ harnesses floating point efficiency
  - $\Rightarrow$ embarrassingly easy parallelization (and fault tolerance)
- Favor **tiled recursive** algorithms
  - $\Rightarrow$ **architecture oblivious vs aware** algorithms [Gustavson 07]
- New pivoting strategies revealing **all rank profile informations**
  - $\Rightarrow$ **tournament pivoting**? [Demmel, Grigori and Xiang 11]
Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- So far, floating point arithmetic delivers best speed
- Medium size arithmetic: RNS
  - harnesses floating point efficiency
  - embarrassingly easy parallelization (and fault tolerance)
- Favor tiled recursive algorithms
  - architecture oblivious vs aware algorithms [Gustavson 07]
- New pivoting strategies revealing all rank profile informations
  - tournament pivoting? [Demmel, Grigori and Xiang 11]
- Seek size-dimension trade-offs, even heuristic ones,
Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction $\triangleright$ algorithm tuning $\triangleright$ building block implementation

- So far, **floating point** arithmetic delivers best speed
- Medium size arithmetic: **RNS**
  - $\rightsquigarrow$ harnesses floating point efficiency
  - $\rightsquigarrow$ embarrassingly easy parallelization (and fault tolerance)
- Favor **tiled recursive** algorithms
  - $\rightsquigarrow$ **architecture oblivious vs aware** algorithms [Gustavson 07]
- New pivoting strategies revealing **all rank profile informations**
  - $\rightsquigarrow$ **tournament pivoting**? [Demmel, Grigori and Xiang 11]
- Seek **size-dimension** trade-offs, even heuristic ones,
- **Recursive tasks** and **coarse grain** parallelization
  - $\rightsquigarrow$ Light weight task workstealing management required
  - $\rightsquigarrow$ Need for an improved recursive **dataflow** scheduling
Perspectives

Large scale distributed exact linear algebra

- reducing communications [Demmel, Grigori and Xiang 11]
- sparse and hybrid [Faugère and Lachartre 10]
## Perspectives

### Large scale distributed exact linear algebra

- reducing communications [Demmel, Grigori and Xiang 11]
- sparse and hybrid [Faugère and Lachartre 10]

### Structured linear algebra

- A lot of action recently [Jeannerod Schost 08], [Chowdhury & Al. 15]
- Combined with recent advances in linear algebra over $K[X]$
- Applications to list decoding
Perspectives

Large scale distributed exact linear algebra

- reducing communications [Demmel, Grigori and Xiang 11]
- sparse and hybrid [Faugère and Lachartre 10]

Structured linear algebra

- A lot of action recently [Jeannerod Schost 08], [Chowdhury & Al. 15]
- Combined with recent advances in linear algebra over $K[X]$
- Applications to list decoding

Symbolic-numeric computation

- High precision floating point linear algebra via exact rational arithmetic and RNS
Perspectives

Large scale distributed exact linear algebra
- reducing communications [Demmel, Grigori and Xiang 11]
- sparse and hybrid [Faugère and Lachartre 10]

Structured linear algebra
- A lot of action recently [Jeannerod Schost 08], [Chowdhury & Al. 15]
- Combined with recent advances in linear algebra over $K[X]$
- Applications to list decoding

Symbolic-numeric computation
- High precision floating point linear algebra via exact rational arithmetic and RNS

Thank you