Shortest Lattice Vector Enumeration on Graphics Cards

Jens Hermans¹  Michael Schneider²  Fréderik Vercauteren¹
Johannes Buchmann²  Bart Preneel¹

¹K.U.Leuven
²TU Darmstadt

SHARCS - 10 September 2009
Why GPU?

(Source: MSI)
CUDA framework

Warning: sales talk

Your own personal supercomputer for < €500.

Nvidia CUDA Framework:

- Run ‘general’ programs on GPU
- More complex operations, data types, branching...
- Recent GPU required
- Theory: 1TFlop (practice: 200 GFlop)
Crypto on GPU

Current applications:

- Ciphers:
  - RSA \(^1\), ECC \(^2\), AES \(^3\)

- Cryptanalysis:
  - Factoring \(^4\)
  - Brute force

Focus: high throughput, not latency

---

\(^1\) Moss, Page, Smart / Szerwinski, Guneysu / Fleissner
\(^2\) Szerwinski, Guneysu
\(^3\) Manavski / Harrison, Waldron
\(^4\) Bernstein, Chen, Cheng, Lange, Yang
1. Introduction

2. Lattices: crash course

3. GPUs

4. The Algorithm

5. Results

6. The Future
Basis matrix $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ with $\mathbf{b}_i \in \mathbb{R}^d$

Lattice: $L(\mathbf{B}) = \{\sum_{i=1}^n x_i \mathbf{b}_i, x_i \in \mathbb{Z}\}$
Shortest Vector Problem (SVP)

- Basis not unique
- Idea: 'good' basis $B$ and 'bad' basis $B'$
- Finding $\lambda_1(L)$ is hard with $B'$
Algorithms for SVP

Shortest vector problem

Compute $\min_{x \in \mathbb{Z}^n} \|Bx\|_2$

SVP algorithms:

- LLL (+variants): approximate solution, polynomial
- BKZ
- ...
- Enum: exact solution, exponential

⇒ This talk: focus on enum.
Optimum $A = \|Bx\|_2^2$ and $x = [1, 0, \ldots, 0]$
Intermediate norm $l_2$ s.t. $l_i \geq l_{i+1}$ (with $l_1 = \|Bx\|_2^2$)
New optimum $A = \|Bx\|_2^2$
Enumeration

$x_1 = \ldots$

$x_2 = \ldots$

$x_{n-1} = \ldots$

$x_n = \ldots$

Cut off branch if $l_i > A$. 

Jens Hermans, Michael Schneider, Frédéric Vercauteren, Johannes Buchmann, Bart Preneel

Shortest Lattice Vector Enumeration on Graphics Cards
Enumeration

\[ x_1 = \ldots \]

\[ x_2 = \ldots \]

\[ x_{n-1} = \ldots \]

\[ x_n = \ldots \]
Enumeration

$x_n = \ldots$

$x_{n-1} = \ldots$

$x_2 = \ldots$

$x_1 = \ldots$
Enumeration

\[ x_n = \ldots \]
\[ x_{n-1} = \ldots \]
\[ x_2 = \ldots \]
\[ x_1 = \ldots \]
1 Introduction

2 Lattices: crash course

3 GPUs

4 The Algorithm

5 Results

6 The Future
Nvidia GTX280:

- 240 cores, scalar processors
- 30 multiprocessors (8 cores each)
- 1.3 GHz
- 1GB Global Memory
- 32 & 64-bit integers, FP
Programming model

(Source: CUDA programming guide)
Memory types

(Source: CUDA programming guide)
Algorithm Flow

\[ x_1 \rightarrow x_\alpha \rightarrow x_n \]

\( \cdots \)

Jens Hermans, Michael Schneider, Frédéric Vercauteren, Johannes Buchmann
**Basic idea**

Input: $B$, $A$, $\alpha$, $n$

1. Compute the Gram-Schmidt decomposition of $b_i$

3. Output: $(x_1, \ldots, x_n)$ with $\|\sum_{i=1}^{n} x_i b_i\| = \lambda_1(L)$
Basic idea

Input: \( B, A, \alpha, n \)

1. Compute the Gram-Schmidt decomposition of \( b_i \)
2. CPU: generate \( x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n] \)
3. 

Output: \((x_1, \ldots, x_n)\) with \( \left\| \sum_{i=1}^n x_i b_i \right\| = \lambda_1(L) \)
Basic idea

Input: \( B, A, \alpha, n \)

1. Compute the Gram-Schmidt decomposition of \( \mathbf{b}_i \)
2. CPU: generate \( \mathbf{x}_i = [0, \ldots, 0, x_\alpha, \ldots, x_n] \)
3. GPU thread: run a sub-enum on \( \mathbf{x}_i \), if new optimum \( \rightarrow \) store in \( \mathbf{x} \)

Output: \((x_1, \ldots, x_n)\) with \( \left\| \sum_{i=1}^{n} x_i \mathbf{b}_i \right\| = \lambda_1(L) \)
Basic idea

Input: $B, A, \alpha, n$

1. Compute the Gram-Schmidt decomposition of $b_i$
2. CPU: generate $x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n]$
3. GPU thread: run a sub-enum on $x_i$, if new optimum $\rightarrow$ store in $x$

Output: $(x_1, \ldots, x_n)$ with $\|\sum_{i=1}^n x_i b_i\| = \lambda_1(L)$

$\Rightarrow$ horrible performance
Early termination...

---

**Input:** $B, A, \alpha, n$

1. Compute the Gram-Schmidt decomposition of $b_i$
2. CPU: generate $x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n]$

---

**Output:** $(x_1, \ldots, x_n)$ with $\|\sum_{i=1}^{n} x_i b_i\| = \lambda_1(L)$
Early termination...

---

**Input:** $B, A, \alpha, n$

1. Compute the Gram-Schmidt decomposition of $b_i$
2. CPU: generate $x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n]$
3. GPU thread:
4. **while there are $x_i$ left. do**
5.   Start enum for a certain $x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n]$
6.   Stop enum after $S$ steps, store the state \{\(l_i, \bar{x}_i, s_i = S\}\)
7. **end**

8
9

**Output:** \((x_1, \ldots, x_n)\) with \(\|\sum_{i=1}^{n} x_i b_i\| = \lambda_1(L)\)
Early termination...

---

**Input:** $B, A, \alpha, n$

1. Compute the Gram-Schmidt decomposition of $b_i$
2. CPU: generate $x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n]$
3. GPU thread:
   4. while there are $x_i$ left. do
      5. Start enum for a certain $x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n]$
      6. Stop enum after $S$ steps, store the state $\{l_i, \bar{x}_i, s_i = S\}$
   7. end
8. CPU: Get enum state $\bar{x}_i = [\bar{x}_1, \ldots, \bar{x}_{\alpha-1}, x_\alpha, \ldots, x_n]$
9. **Output:** $(x_1, \ldots, x_n)$ with $\|\sum_{i=1}^{n} x_i b_i\| = \lambda_1(L)$
Early termination...

**Input:** \( B, A, \alpha, n \)

1. Compute the Gram-Schmidt decomposition of \( b_i \)
2. CPU: generate \( x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n] \)
3. GPU thread:
   4. while there are \( x_i \) left. do
      5. Start enum for a certain \( x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n] \)
      6. Stop enum after \( S \) steps, store the state \( \{l_i, \bar{x}_i, s_i = S\} \)
   7. end
4. CPU: Get enum state \( \bar{x}_i = [\bar{x}_1, \ldots, \bar{x}_{\alpha-1}, x_\alpha, \ldots, x_n] \)
5. CPU: Continue enum if \( S \) was reached

**Output:** \((x_1, \ldots, x_n)\) with \( \| \sum_{i=1}^{n} x_i b_i \| = \lambda_1(L) \)

\[ \implies \text{solves length difference problem, still not so good} \]
**Input:** $B, A, \alpha, n$

1. Compute the Gram-Schmidt decomposition of $b_i$
2. **while** true **do**
3.   CPU: generate some $x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n]$

9.   CPU: Get enum state $\bar{x}_i = [\bar{x}_1, \ldots, \bar{x}_{\alpha-1}, x_\alpha, \ldots, x_n]$
10. **end**

**Output:** $(x_1, \ldots, x_n)$ with $\left\| \sum_{i=1}^{n} x_i b_i \right\| = \lambda_1(L)$
# Iterating

**Input:** $B, A, \alpha, n$

1. Compute the Gram-Schmidt decomposition of $b_i$
2. **while** true **do**
   3. CPU: generate some $x_i = [0, \ldots, 0, x_\alpha, \ldots, x_n]$
   4. GPU thread:
      5. **while** there are $x_i$ left. **do**
         6. Start enum for a certain $x_i$ or continue enum for $\bar{x}_i$
         7. Stop enum after $S$ steps, store the state $\{l_i, \bar{x}_i, s_i = S\}$
      8. **end**
   9. CPU: Get enum state $\bar{x}_i = [\bar{x}_1, \ldots, \bar{x}_{\alpha-1}, x_\alpha, \ldots, x_n]$
10. **end**

**Output:** $(x_1, \ldots, x_n)$ with $\|\sum_{i=1}^n x_i b_i\| = \lambda_1(L)$
GPU Enumeration

$x_\alpha \ldots x_n$

$x_1$

Introduction
Lattices: crash course
GPUs
The Algorithm
Results
The Future

Jens Hermans, Michael Schneider, Frédéric Vercauteren, Johannes Buchmann

Shortest Lattice Vector Enumeration on Graphics Cards
GPU Enumeration

$x_\alpha \ldots x_n$

$x_1$
GPU Enumeration
GPU Enumeration

$x_{\alpha} \ldots x_n$

$x_1$
GPU Enumeration

\[ x_\alpha \ldots x_n \]

\[ x_1 \]
Implementation details

Some facts & figures:

- Dimension 50, 100000 starting vectors → upload & download ~ 20 MB of data to GPU
- CPU top enum: very fast (low dimension)
- GPU runs for > 10 seconds per iteration, iteration overhead is limited
- Share new optimal values among GPU threads
1 Introduction

2 Lattices: crash course

3 GPUs

4 The Algorithm

5 Results

6 The Future
Throughput:
- CPU: around $25 \cdot 10^6$ steps/s
- GPU: up to $100 \cdot 10^6$ steps/s

Throughput on GPU depends on:
- Lattice dimension $n$
- Length of sub-enumerations
- Number of parallel threads, uploaded points...
Results

The chart shows the time [s] taken for the Shortest Lattice Vector Enumeration on Graphics Cards using CUDA and fpLLL algorithms. The x-axis represents different values of n (45, 48, 50, 52) and the y-axis represents time in seconds. The results indicate a comparison of performance for different values of n using CUDA and fpLLL.
<table>
<thead>
<tr>
<th>n</th>
<th>45</th>
<th>48</th>
<th>50</th>
<th>52</th>
<th>54</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpLLL</td>
<td>18.3s</td>
<td>139s</td>
<td>277s</td>
<td>2483s</td>
<td>6960s</td>
</tr>
<tr>
<td>CUDA</td>
<td>20.2s</td>
<td>92s</td>
<td>133s</td>
<td>959s</td>
<td>2599s</td>
</tr>
<tr>
<td></td>
<td>110%</td>
<td>66%</td>
<td>48%</td>
<td>39%</td>
<td>37%</td>
</tr>
</tbody>
</table>

**Table:** Average time needed for enumeration of lattices in each dimension n.
Ideas for the Future

Future:
- Generalize ideas (not specific for gpu’s... clusters?)
- Use full power of CPU (now: idle during gpu-time)
- Gaussian heuristic
The end...

Questions?
Algorithm 1: High-level GPU ENUM Algorithm

Input: \( b_i, A, \alpha, n \)

1. Compute the Gram-Schmidt decomposition of \( b_i \)
2. while true do
3. \[ S = \{ (x_i, \Delta x_i, \Delta^2 x_i, l_i = \alpha, s_i = 0) \} \] ← Top enum: generate at most \text{NUMSTARTPOINTS} - \# \( T \) vectors
4. \[ R = \{ (\bar{x}_i, \Delta x_i, \Delta^2 x_i, l_i, s_i) \} \] ← GPU enumeration, starting from \( S \cup T \)
5. \[ T \leftarrow \{ R_i : s_i \geq S \} \]
6. if \# \( T \) < \text{CPUTHRESHOLD} then
7. Enumerate the starting points in \( T \) on the CPU.
8. Stop
9. end
10. end

Output: \((x_1, \ldots, x_n)\) with \[ \| \sum_{i=1}^n x_i b_i \| = \lambda_1(L) \]