Advanced Lattice Sieving on GPUs, with Tensor Cores

Léo Ducas, Marc Stevens, **Wessel van Woerden** (CWI).

Centrum Wiskunde & Informatica

Overview

- Most NIST PQC finalists (**5/7**) are based on hard lattice problems.

Overview

- Most NIST PQC finalists (**5/7**) are based on hard lattice problems.
- Practical cryptanalysis is important to pick concrete parameters.

Overview

- Most NIST PQC finalists (**5/7**) are based on hard lattice problems.
- Practical cryptanalysis is important to pick concrete parameters.
- Lattice sieving algorithms have the best practical and asymptotic runtime.

Overview

- Most NIST PQC finalists (**5/7**) are based on hard lattice problems.
- Practical cryptanalysis is important to pick concrete parameters.
- Lattice sieving algorithms have the best practical and asymptotic runtime.
- How fit are (different) sieving algorithms for specialized hardware?

Overview

- Most NIST PQC finalists (**5**/**7**) are based on hard lattice problems.
- Practical cryptanalysis is important to pick concrete parameters.
- Lattice sieving algorithms have the best practical and asymptotic runtime.
- How fit are (different) sieving algorithms for specialized hardware?
- Including more advanced sieving techniques.

Overview

- Most NIST PQC finalists (**5/7**) are based on hard lattice problems.
- Practical cryptanalysis is important to pick concrete parameters.
- Lattice sieving algorithms have the best practical and asymptotic runtime.
- How fit are (different) sieving algorithms for specialized hardware?
- Including more advanced sieving techniques.

Contributions

- First GPU implementation using all state-of-the-art sieving techniques.

- Most NIST PQC finalists (**5**/**7**) are based on hard lattice problems.
- Practical cryptanalysis is important to pick concrete parameters.
- Lattice sieving algorithms have the best practical and asymptotic runtime.
- How fit are (different) sieving algorithms for specialized hardware?
- Including more advanced sieving techniques.

Contributions

- First GPU implementation using all state-of-the-art sieving techniques.
- Improves both runtime and energy efficiency by two orders of magnitude.

Overview

- Most NIST PQC finalists (**5/7**) are based on hard lattice problems.
- Practical cryptanalysis is important to pick concrete parameters.
- Lattice sieving algorithms have the best practical and asymptotic runtime.
- How fit are (different) sieving algorithms for specialized hardware?
- Including more advanced sieving techniques.

Contributions

- First GPU implementation using all state-of-the-art sieving techniques.
- Improves both runtime and energy efficiency by two orders of magnitude.
- Significantly improve several lattice problem records.

Overview

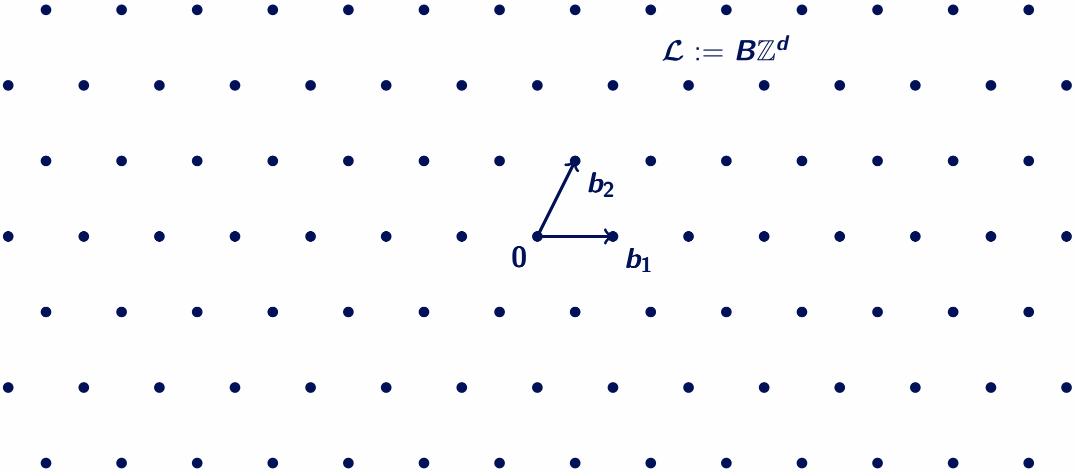
- Most NIST PQC finalists (**5/7**) are based on hard lattice problems.
- Practical cryptanalysis is important to pick concrete parameters.
- Lattice sieving algorithms have the best practical and asymptotic runtime.
- How fit are (different) sieving algorithms for specialized hardware?
- Including more advanced sieving techniques.

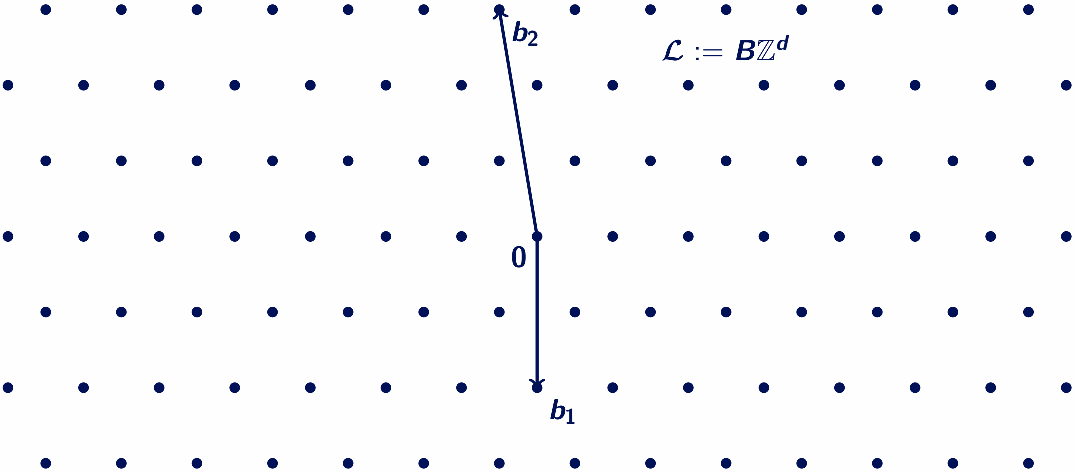
Contributions

- First GPU implementation using all state-of-the-art sieving techniques.
- Improves both runtime and energy efficiency by two orders of magnitude.
- Significantly improve several lattice problem records.
- First optimized implementation of the asymptotic best known sieve [BDGL].

Lattice

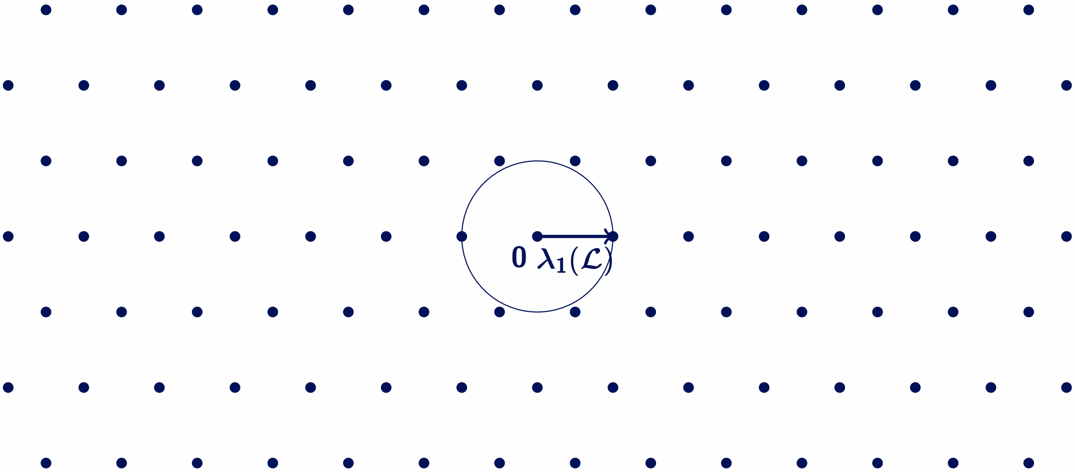
$$\mathcal{L} := \boldsymbol{B}\mathbb{Z}^{\boldsymbol{d}}$$



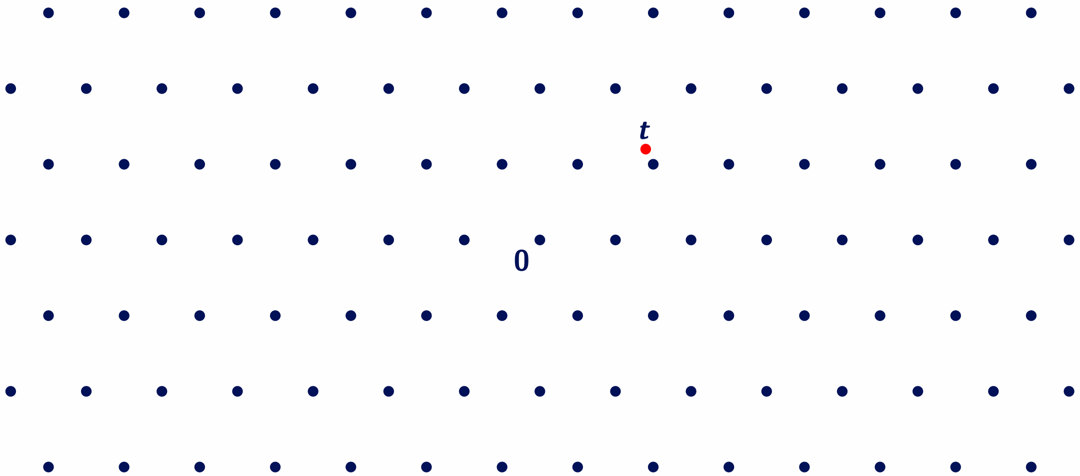


$\mathcal{L} := B\mathbb{Z}^d$

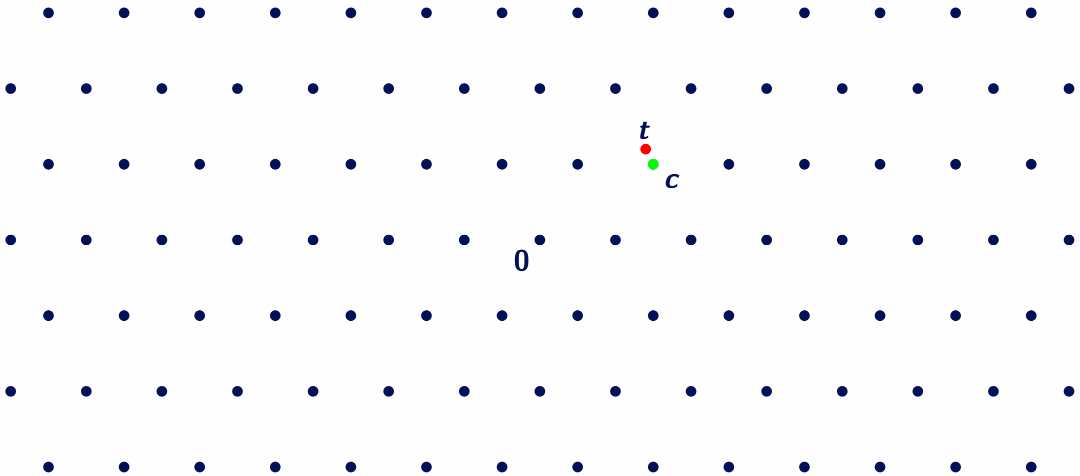
Shortest Vector Problem



Bounded Distance Decoding



Bounded Distance Decoding



- Gives an indication of the concrete hardness of SVP.

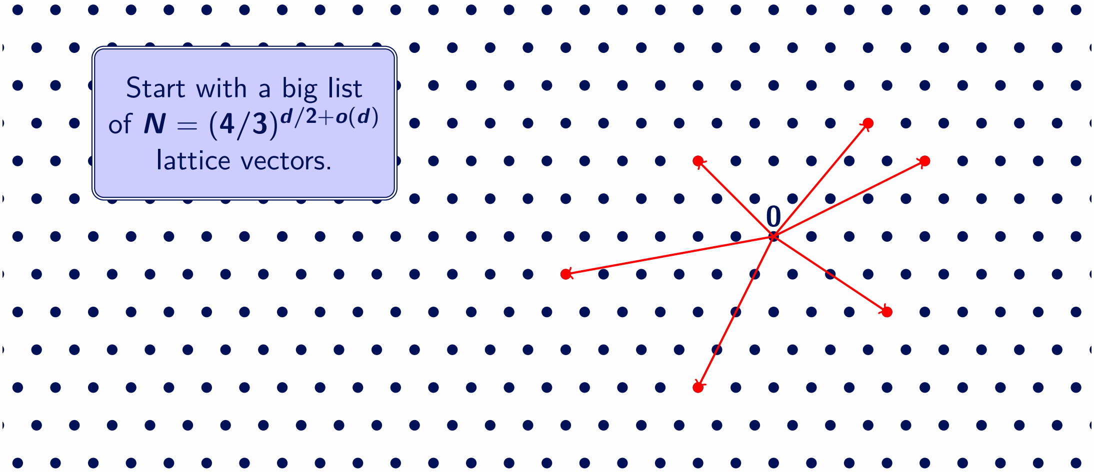
TU Darmstadt Lattice Challenge

- Gives an indication of the <u>concrete</u> hardness of SVP.
- Given: 'Random' $\boldsymbol{d}$-dimensional lattice $\mathcal{L}$ (Goldstein and Mayer)

TU Darmstadt Lattice Challenge

- Gives an indication of the concrete hardness of SVP.
- Given: 'Random' $\boldsymbol{d}$-dimensional lattice $\boldsymbol{\mathcal{L}}$ (Goldstein and Mayer)
- Goal: Find a $\boldsymbol{v} \in \boldsymbol{\mathcal{L}}$ s.t.

$$\|\boldsymbol{v}\| \leq \mathbf{1.05} \cdot \mathrm{GH}(\boldsymbol{\mathcal{L}}) \approx \mathbf{1.05} \cdot \boldsymbol{\lambda_1}(\boldsymbol{\mathcal{L}}).$$

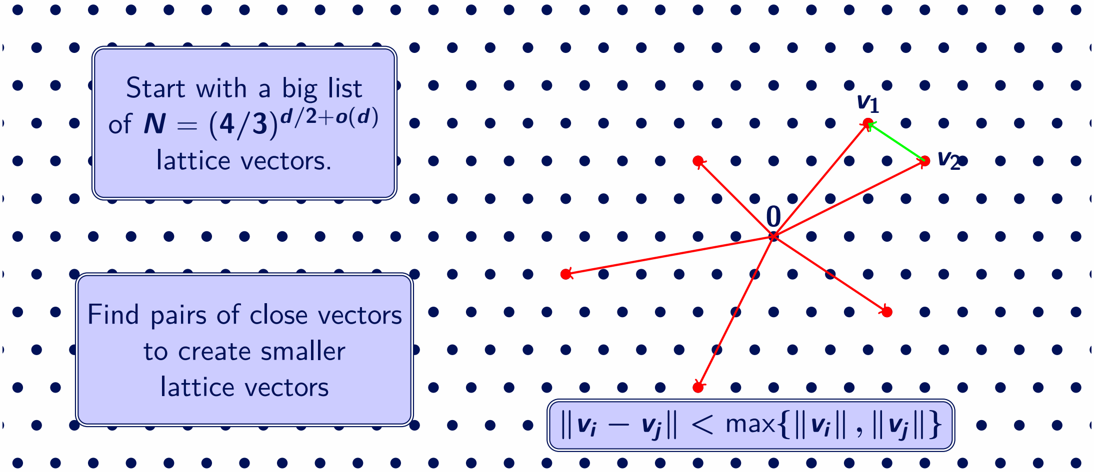


Start with a big list of $N = (4/3)^{d/2+o(d)}$ lattice vectors.

Lattice Sieving

Start with a big list of $N = (4/3)^{d/2+o(d)}$ lattice vectors.

Find pairs of close vectors to create smaller lattice vectors

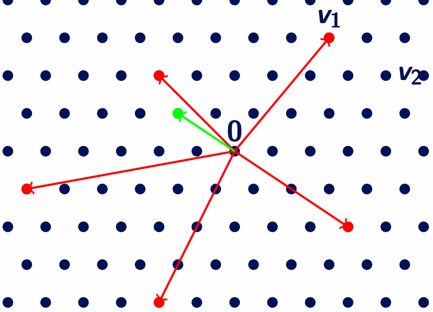


$\|v_i - v_j\| < \max\{\|v_i\|, \|v_j\|\}$

Lattice Sieving

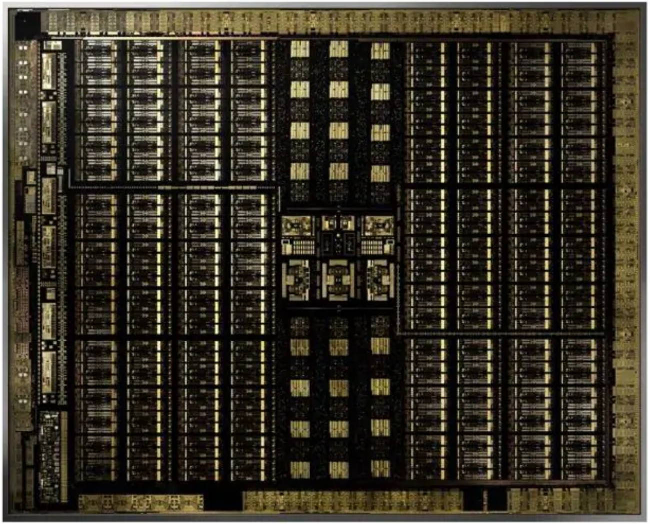
Start with a big list of $N = (4/3)^{d/2+o(d)}$ lattice vectors.

Find pairs of close vectors to create smaller lattice vectors



$\|v_i - v_j\| < \max\{\|v_i\|, \|v_j\|\}$

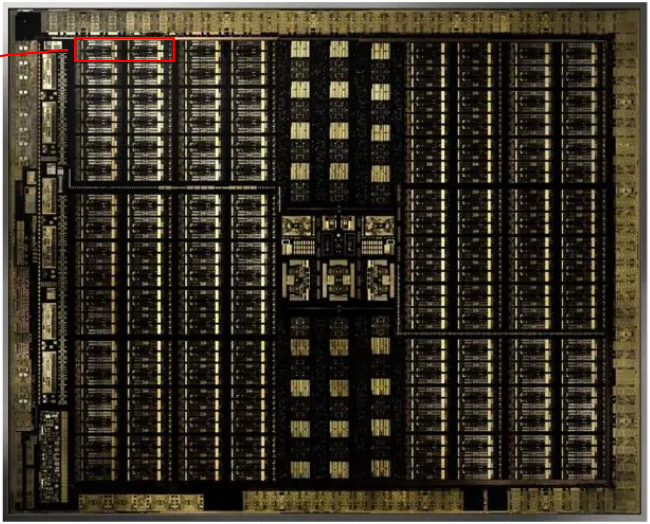
Graphics Processing Unit (GPU)



Graphics Processing Unit (GPU)

64 FP32 cores
64 INT32 cores
8 Tensor cores.

Thousands of cores.

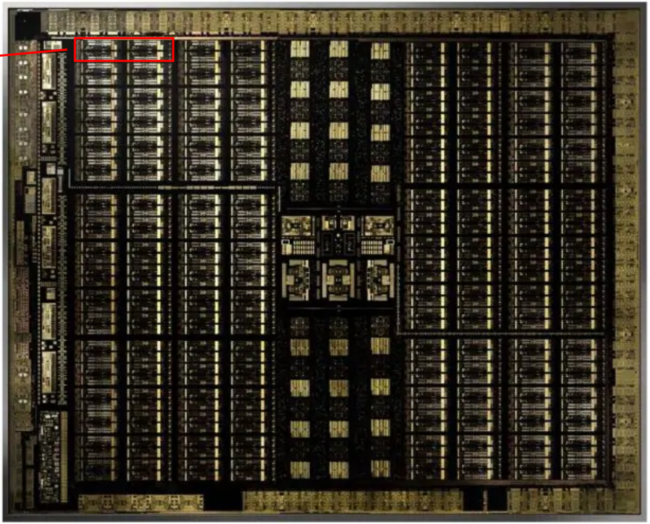


Graphics Processing Unit (GPU)

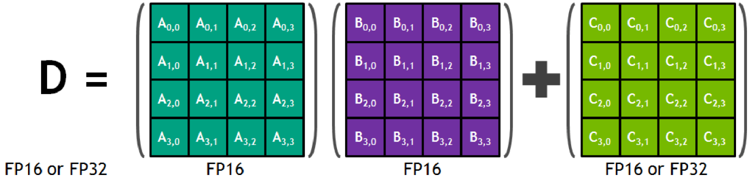
64 FP32 cores
64 INT32 cores
8 Tensor cores.

Thousands of cores.

Per **32** cores:
Single Instruction
Multiple Data

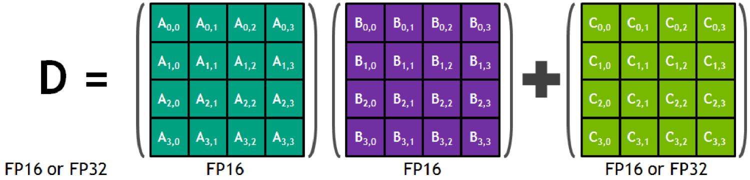


Tensor Cores



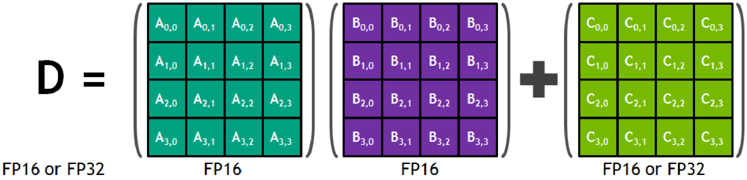
- Very efficient (low precision) matrix multiplication.

Tensor Cores



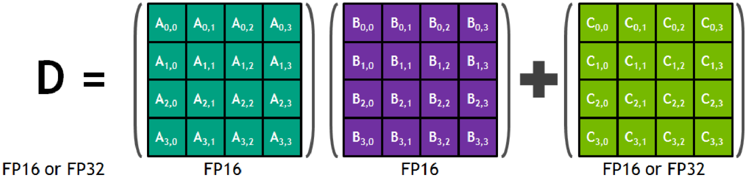
- Very efficient (low precision) matrix multiplication.
- **16**-bit precision is good enough.

Tensor Cores



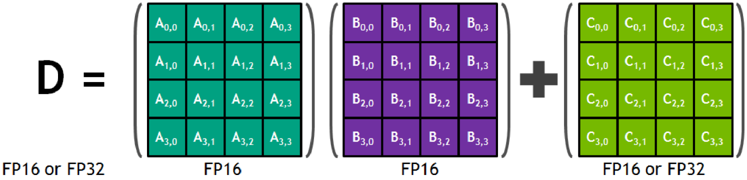
- Very efficient (low precision) matrix multiplication.
- **16**-bit precision is good enough.
- Up to **108** 16-bit **Tflops**! [2018 model we used]

Tensor Cores



- Very efficient (low precision) matrix multiplication.
- **16**-bit precision is good enough.
- Up to **108** 16-bit **Tflops**! [2018 model we used]
- Newest model $>$ **300** 16-bit **Tflops**.

Tensor Cores



- Very efficient (low precision) matrix multiplication.
- **16**-bit precision is good enough.
- Up to **108** 16-bit **Tflops**! [2018 model we used]
- Newest model > **300** 16-bit **Tflops**.
- The current best CPU would reach at most ≈ **5** 16-bit **Tflops**.

GPU: pros and cons

Cons

Not versatile.

Pros

GPU: pros and cons

Cons

Not versatile.

'External' device.

Pros

GPU: pros and cons

Cons

Not versatile.

'External' device.

Memory bottlenecks often limit actual performance.

Pros

GPU: pros and cons

Cons

Not versatile.

'External' device.

Memory bottlenecks often limit actual performance.

Hard to adapt algorithms.

Pros

GPU: pros and cons

Cons

Not versatile.

'External' device.

Memory bottlenecks often limit actual performance.

Hard to adapt algorithms.

Pros

Incredible performance.

Cons

Not versatile.

'External' device.

Memory bottlenecks often limit actual performance.

Hard to adapt algorithms.

Pros

Incredible performance.

Energy efficient.

G6K, Albrecht et al. 2019

- Open source sieving framework/implementation.

G6K, Albrecht et al. 2019

- Open source sieving framework/implementation.
- Advanced sieving: Combines all state-of-the-art results/'tricks'.

G6K, Albrecht et al. 2019

- Open source sieving framework/implementation.
- Advanced sieving: Combines all state-of-the-art results/'tricks'.
- Fully parallel (CPU, single machine).

G6K, Albrecht et al. 2019

- Open source sieving framework/implementation.
- Advanced sieving: Combines all state-of-the-art results/'tricks'.
- Fully parallel (CPU, single machine).
- SVP record at dimension **155** in $\pm$**14** days

G6K, Albrecht et al. 2019

- Open source sieving framework/implementation.
- Advanced sieving: Combines all state-of-the-art results/‘tricks’.
- Fully parallel (CPU, single machine).
- SVP record at dimension **155** in $\pm$**14** days
  - **4** $\times$ **18** cpu cores.

G6K, Albrecht et al. 2019

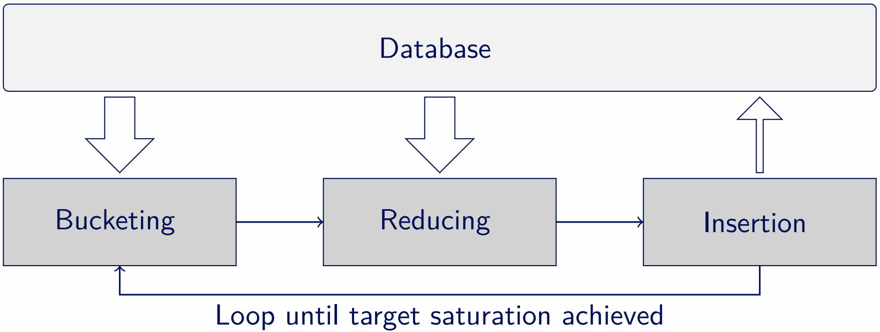
- Open source sieving framework/implementation.
- Advanced sieving: Combines all state-of-the-art results/'tricks'.
- Fully parallel (CPU, single machine).
- SVP record at dimension **155** in $\pm$**14** days
  - **4** $\times$ **18** cpu cores.
  - $\approx$ **256*GiB*** memory.

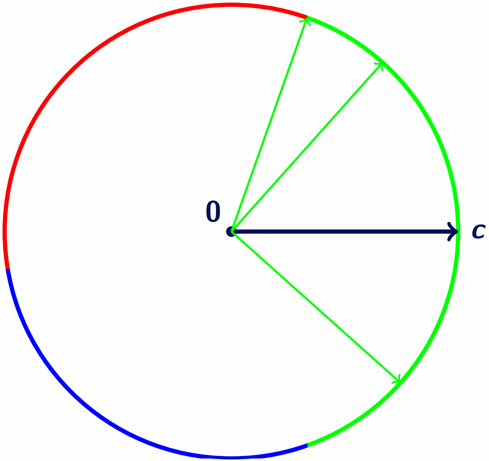
G6K, Albrecht et al. 2019

- Open source sieving framework/implementation.
- Advanced sieving: Combines all state-of-the-art results/'tricks'.
- Fully parallel (CPU, single machine).
- SVP record at dimension **155** in $\pm$**14** days
  - **4** $\times$ **18** cpu cores.
  - $\approx$ **256*GiB*** memory.
- Enumeration: dimension **152** using **800.000** core hours!.

Advanced Sieving on GPUs

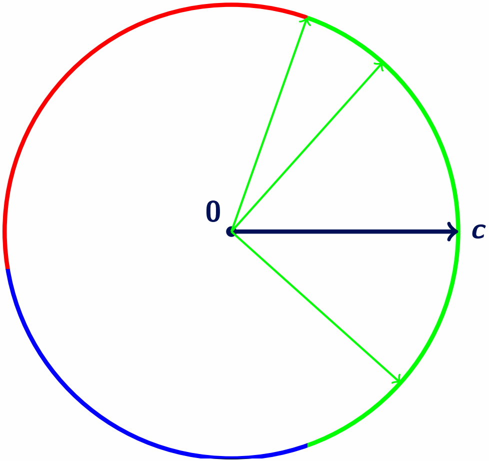
Sieving Process





Partition the sphere.

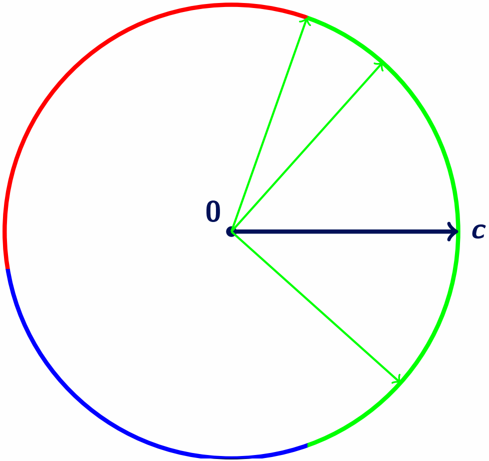
Bucketing



Partition the sphere.

Only check all pairs <u>within</u> each bucket.

Bucketing



Partition the sphere.

Only check all pairs <u>within</u> each bucket.

Increases reduction probability per pair.

Bucketing

- No bucketing: time $N^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.

Bucketing

- No bucketing: time $\boldsymbol{N}^2 = (\mathbf{4/3})^{\boldsymbol{d}+\boldsymbol{o}(\boldsymbol{d})} = \mathbf{2}^{\mathbf{0.415}\boldsymbol{d}+\boldsymbol{o}(\boldsymbol{d})}$.
- [BGJ1] Random spherical cones.

Bucketing

- No bucketing: time $\boldsymbol{N}^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.
- [BGJ1] Random spherical cones.
  - $\sqrt{\boldsymbol{N}}$ buckets of size $\sqrt{\boldsymbol{N}}$.

Bucketing

- No bucketing: time $N^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.
- [BGJ1] Random spherical cones.
  - $\sqrt{N}$ buckets of size $\sqrt{N}$.
  - time $2^{0.349d+o(d)}$.

Bucketing

- No bucketing: time $N^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.
- [BGJ1] Random spherical cones.
  - $\sqrt{N}$ buckets of size $\sqrt{N}$.
  - time $2^{0.349d+o(d)}$.
  - Trick: Replace random directions with random lattice vectors.

Bucketing

- No bucketing: time $N^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.
- [BGJ1] Random spherical cones.
  - $\sqrt{N}$ buckets of size $\sqrt{N}$.
  - time $2^{0.349d+o(d)}$.
  - Trick: Replace random directions with random lattice vectors.
  - Compute inner products with lattice vectors: **Tensor cores!**

Bucketing

- No bucketing: time $N^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.
- [BGJ1] Random spherical cones.
  - $\sqrt{N}$ buckets of size $\sqrt{N}$.
  - time $2^{0.349d+o(d)}$.
  - Trick: Replace random directions with random lattice vectors.
  - Compute inner products with lattice vectors: **Tensor cores!**
- [BDGL] Structured spherical cones (product code).

Bucketing

- No bucketing: time $N^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.
- [BGJ1] Random spherical cones.
  - $\sqrt{N}$ buckets of size $\sqrt{N}$.
  - time $2^{0.349d+o(d)}$.
  - Trick: Replace random directions with random lattice vectors.
  - Compute inner products with lattice vectors: **Tensor cores!**
- [BDGL] Structured spherical cones (product code).
  - $N^{k/(k+1)}$ buckets of size $N^{1/(k+1)}$ $\leftarrow$ A lot of **small buckets!**

Bucketing

- No bucketing: time $N^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.
- [BGJ1] Random spherical cones.
  - $\sqrt{N}$ buckets of size $\sqrt{N}$.
  - time $2^{0.349d+o(d)}$.
  - Trick: Replace random directions with random lattice vectors.
  - Compute inner products with lattice vectors: **Tensor cores!**
- [BDGL] Structured spherical cones (product code).
  - $N^{k/(k+1)}$ buckets of size $N^{1/(k+1)}$ $\leftarrow$ A lot of **small buckets!**
  - time $2^{0.292d+o(d)}$ for $k = \theta(\log n)$.

Bucketing

- No bucketing: time $N^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.
- [BGJ1] Random spherical cones.
  - $\sqrt{N}$ buckets of size $\sqrt{N}$.
  - time $2^{0.349d+o(d)}$.
  - Trick: Replace random directions with random lattice vectors.
  - Compute inner products with lattice vectors: **Tensor cores!**
- [BDGL] Structured spherical cones (product code).
  - $N^{k/(k+1)}$ buckets of size $N^{1/(k+1)}$ $\leftarrow$ A lot of **small buckets!**
  - time $2^{0.292d+o(d)}$ for $k = \theta(\log n)$.
  - Trick: Implicit directions using permutations and Hadamard transform.

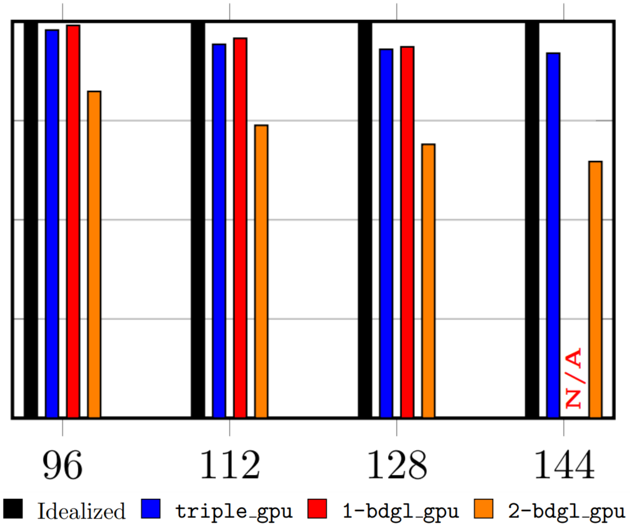
Bucketing

- No bucketing: time $N^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.
- [BGJ1] Random spherical cones.
  - $\sqrt{N}$ buckets of size $\sqrt{N}$.
  - time $2^{0.349d+o(d)}$.
  - Trick: Replace random directions with random lattice vectors.
  - Compute inner products with lattice vectors: **Tensor cores!**
- [BDGL] Structured spherical cones (product code).
  - $N^{k/(k+1)}$ buckets of size $N^{1/(k+1)}$ $\leftarrow$ A lot of **small buckets!**
  - time $2^{0.292d+o(d)}$ for $k = \theta(\log n)$.
  - Trick: Implicit directions using permutations and Hadamard transform.
  - Suitable for CPU (AVX2) and GPU.

Bucketing

- No bucketing: time $N^2 = (4/3)^{d+o(d)} = 2^{0.415d+o(d)}$.
- [BGJ1] Random spherical cones.
  - $\sqrt{N}$ buckets of size $\sqrt{N}$.
  - time $2^{0.349d+o(d)}$.
  - Trick: Replace random directions with random lattice vectors.
  - Compute inner products with lattice vectors: **Tensor cores!**
- [BDGL] Structured spherical cones (product code).
  - $N^{k/(k+1)}$ buckets of size $N^{1/(k+1)}$ $\leftarrow$ A lot of **small buckets!**
  - time $2^{0.292d+o(d)}$ for $k = \theta(\log n)$.
  - Trick: Implicit directions using permutations and Hadamard transform.
  - Suitable for CPU (AVX2) and GPU.
  - AVX2 CPU implementation merged into G6K, fastest CPU sieve.

Bucketing Quality



Reducing

- For each pair $\boldsymbol{v}, \boldsymbol{w}$ in a bucket check if $\|\boldsymbol{v} \pm \boldsymbol{w}\| < \boldsymbol{C}$.

Reducing

- For each pair $\boldsymbol{v}, \boldsymbol{w}$ in a bucket check if $\|\boldsymbol{v} \pm \boldsymbol{w}\| < \boldsymbol{C}$.
- $\|\boldsymbol{v} \pm \boldsymbol{w}\|^2 = \|\boldsymbol{v}\|^2 + \|\boldsymbol{w}\|^2 \pm 2\langle \boldsymbol{v}, \boldsymbol{w} \rangle$.

Reducing

- For each pair $\boldsymbol{v}, \boldsymbol{w}$ in a bucket check if $\|\boldsymbol{v} \pm \boldsymbol{w}\| < \boldsymbol{C}$.
- $\|\boldsymbol{v} \pm \boldsymbol{w}\|^2 = \|\boldsymbol{v}\|^2 + \|\boldsymbol{w}\|^2 \pm 2\langle \boldsymbol{v}, \boldsymbol{w} \rangle$.
- Need to compute pairwise inner products $\langle \boldsymbol{v}, \boldsymbol{w} \rangle$: **Tensor cores!**.

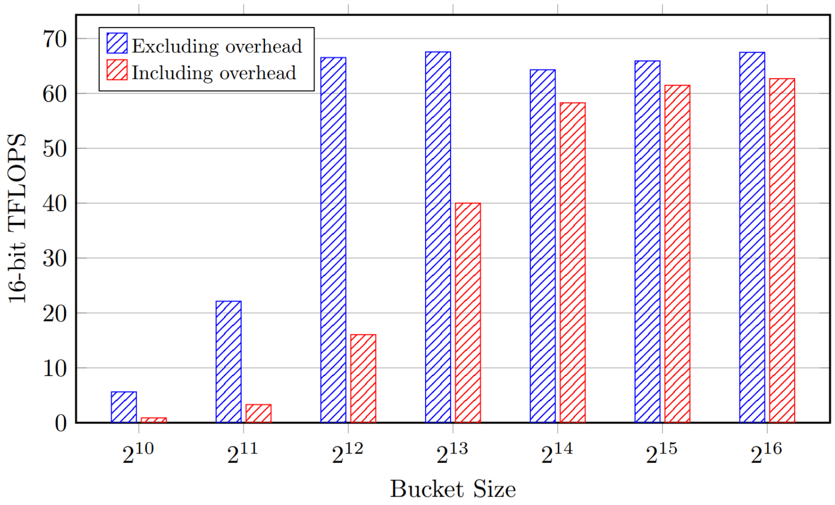
Reducing

- For each pair $\boldsymbol{v}, \boldsymbol{w}$ in a bucket check if $\|\boldsymbol{v} \pm \boldsymbol{w}\| < \boldsymbol{C}$.
- $\|\boldsymbol{v} \pm \boldsymbol{w}\|^2 = \|\boldsymbol{v}\|^2 + \|\boldsymbol{w}\|^2 \pm 2\langle \boldsymbol{v}, \boldsymbol{w} \rangle$.
- Need to compute pairwise inner products $\langle \boldsymbol{v}, \boldsymbol{w} \rangle$: **Tensor cores!**.
- Sparse output: only return successful pairs.

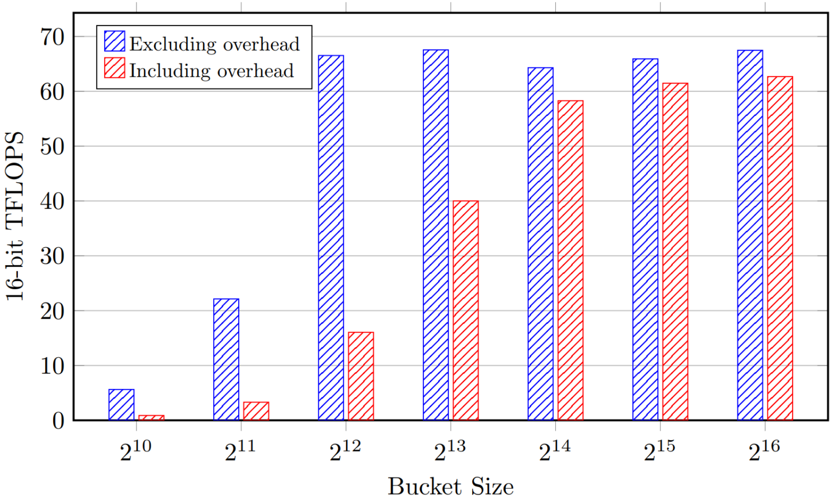
Reducing

- For each pair $\boldsymbol{v}, \boldsymbol{w}$ in a bucket check if $\|\boldsymbol{v} \pm \boldsymbol{w}\| < \boldsymbol{C}$.
- $\|\boldsymbol{v} \pm \boldsymbol{w}\|^2 = \|\boldsymbol{v}\|^2 + \|\boldsymbol{w}\|^2 \pm 2\langle \boldsymbol{v}, \boldsymbol{w} \rangle$.
- Need to compute pairwise inner products $\langle \boldsymbol{v}, \boldsymbol{w} \rangle$: **Tensor cores!**.
- Sparse output: only return successful pairs.
- FLOP: $\boldsymbol{O}(\boldsymbol{d} \cdot \boldsymbol{B}^2)$, data: $\boldsymbol{O}(\boldsymbol{d} \cdot \boldsymbol{B})$, ratio improves for larger bucket size $\boldsymbol{B}$.

Amortizing data throughput

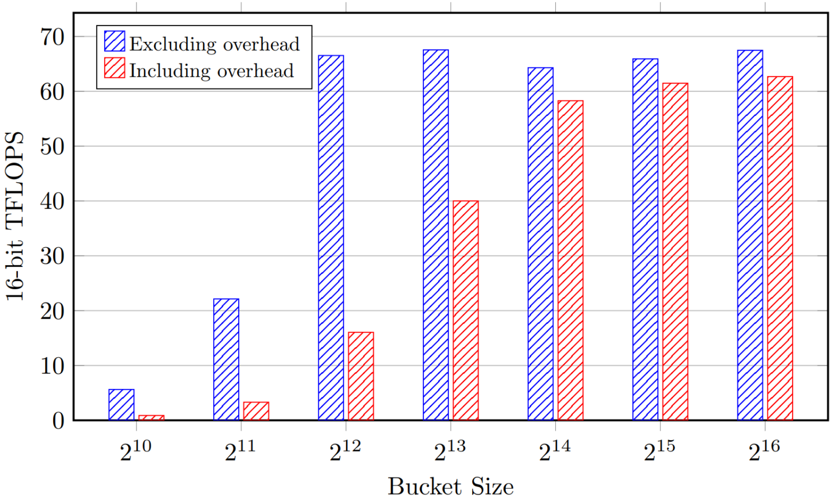


Amortizing data throughput



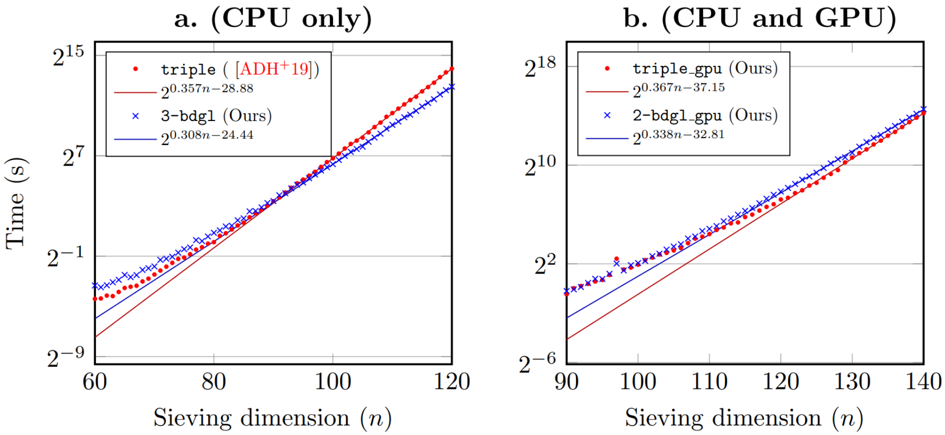
- Small buckets are memory bound.

Amortizing data throughput



- Small buckets are memory bound.
- Large buckets to reach optimal performance.

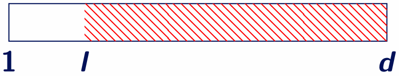
BGJ1 vs BDGL



a. (CPU only)
triple ([ADH+19])
$2^{0.357n-28.88}$
3-bdgl (Ours)
$2^{0.308n-24.44}$
Time (s)
2^{15}
2^{7}
2^{-1}
2^{-9}
60
80
100
120
Sieving dimension (n)
b. (CPU and GPU)
triple_gpu (Ours)
$2^{0.367n-37.15}$
2-bdgl_gpu (Ours)
$2^{0.338n-32.81}$
2^{18}
2^{10}
2^{2}
2^{-6}
90
100
110
120
130
140
Sieving dimension (n)

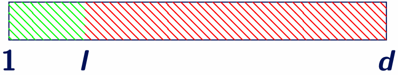
Dimensions for Free [Duc18]

- Sieve in a projected sublattice $\pi_l(\mathcal{L})$ (projected away from first l basis vectors).



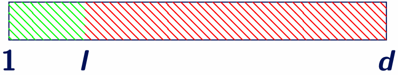
Dimensions for Free [Duc18]

- Sieve in a projected sublattice $\pi_l(\mathcal{L})$ (projected away from first l basis vectors).
- Lift the database back to the full lattice [Babai lifting].



Dimensions for Free [Duc18]

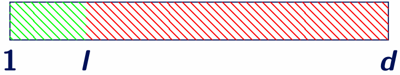
- Sieve in a projected sublattice $\pi_l(\mathcal{L})$ (projected away from first l basis vectors).
- Lift the database back to the full lattice [Babai lifting].



- Finds the shortest vector for $l = O\left(\frac{d}{\log(d)}\right)$.

Dimensions for Free [Duc18]

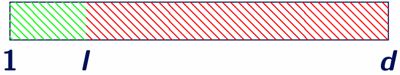
- Sieve in a projected sublattice $\pi_l(\mathcal{L})$ (projected away from first l basis vectors).
- Lift the database back to the full lattice [Babai lifting].



- Finds the shortest vector for $l = O\left(\frac{d}{\log(d)}\right)$.
- Progressive sieving: decrease l step-by-step.

Dimensions for Free [Duc18]

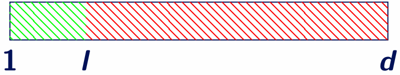
- Sieve in a projected sublattice $\pi_l(\mathcal{L})$ (projected away from first l basis vectors).
- Lift the database back to the full lattice [Babai lifting].



- Finds the shortest vector for $l = O\left(\frac{d}{\log(d)}\right)$.
- Progressive sieving: decrease l step-by-step.
- On the fly lifting: lift any shortish vector we encounter.

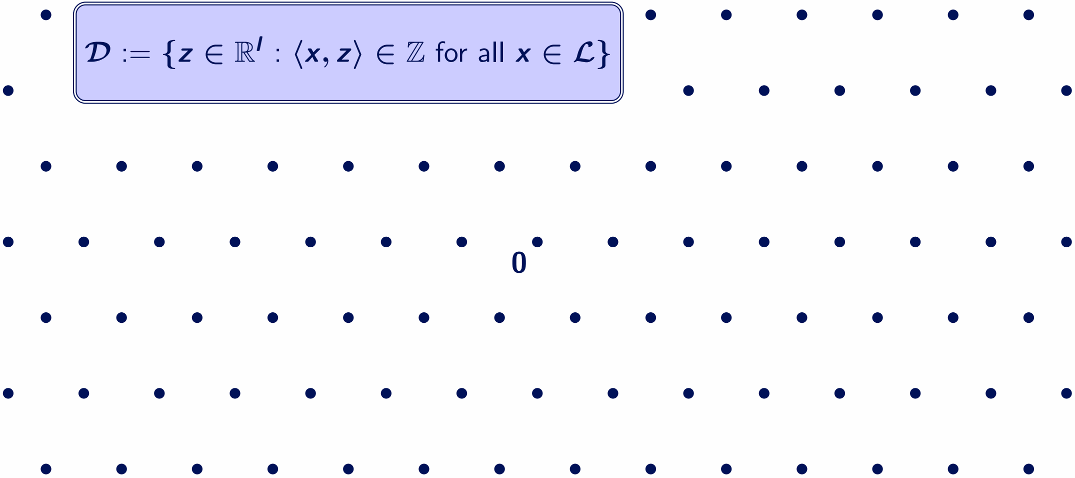
Dimensions for Free [Duc18]

- Sieve in a projected sublattice $\pi_l(\mathcal{L})$ (projected away from first l basis vectors).
- Lift the database back to the full lattice [Babai lifting].



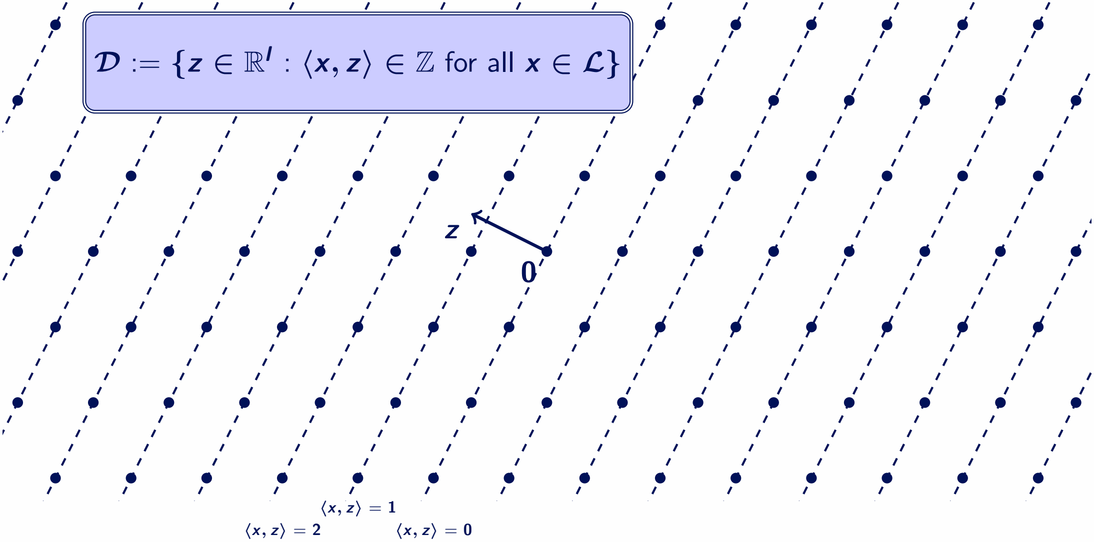
- Finds the shortest vector for $l = O\left(\frac{d}{\log(d)}\right)$.
- Progressive sieving: decrease l step-by-step.
- On the fly lifting: lift any shortish vector we encounter.
- Can we efficiently detect if $v_i - v_j$ might lift to a short vector [BDD problem]?

$$\mathcal{D} := \{\boldsymbol{z} \in \mathbb{R}^{\boldsymbol{l}} : \langle \boldsymbol{x}, \boldsymbol{z} \rangle \in \mathbb{Z} \text{ for all } \boldsymbol{x} \in \mathcal{L}\}$$



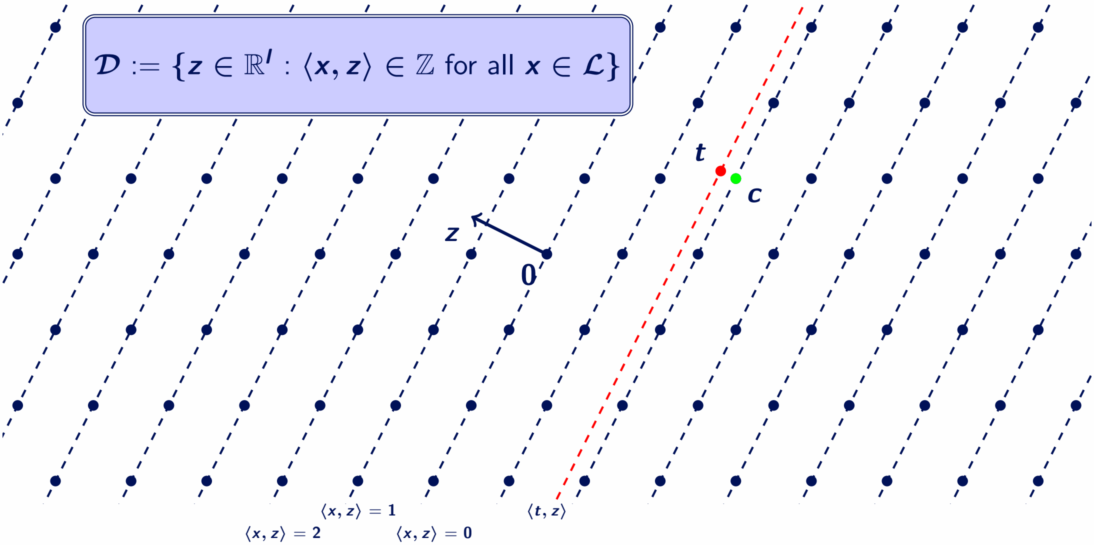
Dual Hash

$$\mathcal{D} := \{ \boldsymbol{z} \in \mathbb{R}^{\boldsymbol{l}} : \langle \boldsymbol{x}, \boldsymbol{z} \rangle \in \mathbb{Z} \text{ for all } \boldsymbol{x} \in \mathcal{L} \}$$



Dual Hash

$$\mathcal{D} := \{ \boldsymbol{z} \in \mathbb{R}^{\boldsymbol{l}} : \langle \boldsymbol{x}, \boldsymbol{z} \rangle \in \mathbb{Z} \text{ for all } \boldsymbol{x} \in \mathcal{L} \}$$



Dual Hash

- For short dual vectors $\boldsymbol{z_1}, \ldots, \boldsymbol{z_k} \in \mathcal{D}$ we define the **dual hash**

$$\mathcal{H}(\boldsymbol{t}) := (\langle \boldsymbol{t}, \boldsymbol{z_i} \rangle)_{\boldsymbol{i}}.$$

Dual Hash

- For short dual vectors $\boldsymbol{z_1}, \ldots, \boldsymbol{z_k} \in \mathcal{D}$ we define the **dual hash**

$$\mathcal{H}(\boldsymbol{t}) := (\langle \boldsymbol{t}, \boldsymbol{z_i} \rangle)_{\boldsymbol{i}}.$$

- If $\mathrm{dist}(\mathcal{L}, \boldsymbol{t})$ is small, then $\mathrm{dist}(\mathbb{Z}^{\boldsymbol{k}}, \mathcal{H}(\boldsymbol{t}))$ is small.

Dual Hash

- For short dual vectors $\boldsymbol{z_1}, \ldots, \boldsymbol{z_k} \in \mathcal{D}$ we define the **dual hash**

$$\mathcal{H}(\boldsymbol{t}) := (\langle \boldsymbol{t}, \boldsymbol{z_i} \rangle)_i.$$

- If $\mathrm{dist}(\mathcal{L}, \boldsymbol{t})$ is small, then $\mathrm{dist}(\mathbb{Z}^k, \mathcal{H}(\boldsymbol{t}))$ is small.
- For $\boldsymbol{l} = \mathbf{16}$, $\boldsymbol{k} = \mathbf{48}$ seems enough for a strong correlation.

Dual Hash

- For short dual vectors $\boldsymbol{z_1}, \ldots, \boldsymbol{z_k} \in \mathcal{D}$ we define the **dual hash**

$$\mathcal{H}(\boldsymbol{t}) := (\langle \boldsymbol{t}, \boldsymbol{z_i} \rangle)_i.$$

- If $\mathrm{dist}(\mathcal{L}, \boldsymbol{t})$ is small, then $\mathrm{dist}(\mathbb{Z}^k, \mathcal{H}(\boldsymbol{t}))$ is small.
- For $\boldsymbol{l} = \mathbf{16}$, $\boldsymbol{k} = \mathbf{48}$ seems enough for a strong correlation.
- $\boldsymbol{H}(\boldsymbol{t_i} - \boldsymbol{t_j}) = \boldsymbol{H}(\boldsymbol{t_i}) - \boldsymbol{H}(\boldsymbol{t_j})$, so $\boldsymbol{O}(\boldsymbol{k})$ operations per pair.

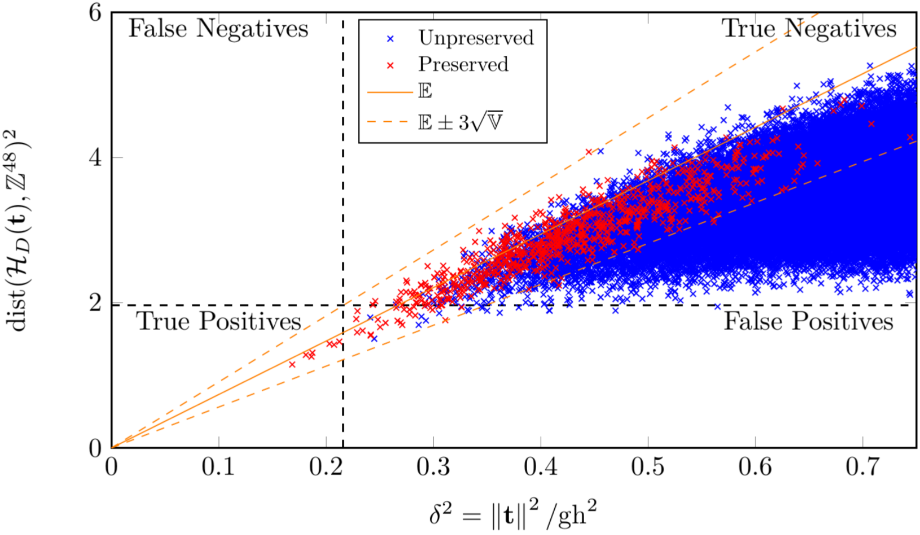
Dual Hash

- For short dual vectors $\boldsymbol{z_1}, \ldots, \boldsymbol{z_k} \in \mathcal{D}$ we define the **dual hash**

$$\mathcal{H}(\boldsymbol{t}) := (\langle \boldsymbol{t}, \boldsymbol{z_i} \rangle)_i.$$

- If $\mathrm{dist}(\mathcal{L}, \boldsymbol{t})$ is small, then $\mathrm{dist}(\mathbb{Z}^k, \mathcal{H}(\boldsymbol{t}))$ is small.
- For $\boldsymbol{l} = \mathbf{16}$, $\boldsymbol{k} = \mathbf{48}$ seems enough for a strong correlation.
- $\boldsymbol{H}(\boldsymbol{t_i} - \boldsymbol{t_j}) = \boldsymbol{H}(\boldsymbol{t_i}) - \boldsymbol{H}(\boldsymbol{t_j})$, so $\boldsymbol{O}(\boldsymbol{k})$ operations per pair.
- Suitable for GPUs.

Dual Hash



False Negatives
True Negatives
True Positives
False Positives
Unpreserved
Preserved
$\mathbb{E}$
$\mathbb{E} \pm 3\sqrt{\mathbb{V}}$
6
4
2
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
$\mathrm{dist}(\mathcal{H}_D(\mathbf{t}), \mathbb{Z}^{48})^2$
$\delta^2 = \|\mathbf{t}\|^2 / \mathrm{gh}^2$

Saving Memory

- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} \boldsymbol{x_i} \boldsymbol{b_i}$:

Saving Memory

- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} \boldsymbol{x_i} \boldsymbol{b_i}$:
  - $[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d]$. $(\mathbf{2}\boldsymbol{d})$

Saving Memory

- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} \boldsymbol{x_i} \boldsymbol{b_i}$:
  - $[\boldsymbol{x}_1, \ldots, \boldsymbol{x_d}]$. $(2\boldsymbol{d})$
  - $[\langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_1 \rangle / \left\| \tilde{\boldsymbol{b}}_1 \right\|, \ldots, \langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_{\boldsymbol{d}} \rangle / \left\| \tilde{\boldsymbol{b}}_{\boldsymbol{d}} \right\|]$. $(4\boldsymbol{d})$

Saving Memory

- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} \boldsymbol{x_i b_i}$:
  - $[\boldsymbol{x}_1, \ldots, \boldsymbol{x_d}]$. $(\boldsymbol{2d})$
  - $[\langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_1 \rangle / \left\| \tilde{\boldsymbol{b}}_1 \right\|, \ldots, \langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_{\boldsymbol{d}} \rangle / \left\| \tilde{\boldsymbol{b}}_{\boldsymbol{d}} \right\|]$. $(\boldsymbol{4d})$
  - $\|\boldsymbol{y}\|^2$. $(\boldsymbol{8})$

Saving Memory

- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} \boldsymbol{x_i} \boldsymbol{b_i}$:
  - $[\boldsymbol{x}_1, \ldots, \boldsymbol{x_d}]$. $(\mathbf{2}\boldsymbol{d})$
  - $[\langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_1 \rangle / \left\| \tilde{\boldsymbol{b}}_1 \right\|, \ldots, \langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_{\boldsymbol{d}} \rangle / \left\| \tilde{\boldsymbol{b}}_{\boldsymbol{d}} \right\|]$. $(\mathbf{4}\boldsymbol{d})$
  - $\|\boldsymbol{y}\|^2$. $(\mathbf{8})$
  - Unique Identifier $(\mathbf{8})$

Saving Memory

- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} x_i \boldsymbol{b_i}$:
  - $[x_1, \ldots, x_d]$. $(\mathbf{2}d)$
  - $[\langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_1 \rangle / \left\| \tilde{\boldsymbol{b}}_1 \right\|, \ldots, \langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_d \rangle / \left\| \tilde{\boldsymbol{b}}_d \right\|]$. $(\mathbf{4}d)$
  - $\|\boldsymbol{y}\|^2$. $(\mathbf{8})$
  - Unique Identifier $(\mathbf{8})$
  - Lift target $\boldsymbol{t}$. $(\mathbf{4}l)$

Saving Memory

- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} x_i \boldsymbol{b_i}$:
  - $[x_1, \ldots, x_d]$. $(\mathbf{2d})$
  - $[\langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_1 \rangle / \left\| \tilde{\boldsymbol{b}}_1 \right\|, \ldots, \langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_d \rangle / \left\| \tilde{\boldsymbol{b}}_d \right\|]$. $(\mathbf{4d})$
  - $\|\boldsymbol{y}\|^2$. $(\mathbf{8})$
  - Unique Identifier $(\mathbf{8})$
  - Lift target $\boldsymbol{t}$. $(\mathbf{4l})$
  - Dual Hash $(\mathbf{4l})$

Saving Memory

- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} x_i \boldsymbol{b}_i$:
  - $[x_1, \ldots, x_d]$. (**2d**)
  - $[\langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_1 \rangle / \left\| \tilde{\boldsymbol{b}}_1 \right\|, \ldots, \langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_d \rangle / \left\| \tilde{\boldsymbol{b}}_d \right\|]$. (**4$d$**)
  - $\|\boldsymbol{y}\|^2$. (**8**)
  - Unique Identifier (**8**)
  - Lift target $\boldsymbol{t}$. (**4l**)
  - Dual Hash (**4l**)
  - Popcount (**d/4**)

Saving Memory

- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} \boldsymbol{x_i} \boldsymbol{b_i}$:
  - $[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d]$. $(\mathbf{2}\boldsymbol{d})$
  - $[\langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_1 \rangle / \left\| \tilde{\boldsymbol{b}}_1 \right\|, \ldots, \langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_d \rangle / \left\| \tilde{\boldsymbol{b}}_d \right\|]$. $(\mathbf{4}\boldsymbol{d})$
  - $\|\boldsymbol{y}\|^2$. $(\mathbf{8})$
  - Unique Identifier $(\mathbf{8})$
  - Lift target $\boldsymbol{t}$. $(\mathbf{4}\boldsymbol{l})$
  - Dual Hash $(\mathbf{4}\boldsymbol{l})$
  - Popcount $(\boldsymbol{d}/\mathbf{4})$
- Remove all except $\boldsymbol{x}$, $\|\boldsymbol{y}\|^2$ and unique identifier.

Saving Memory

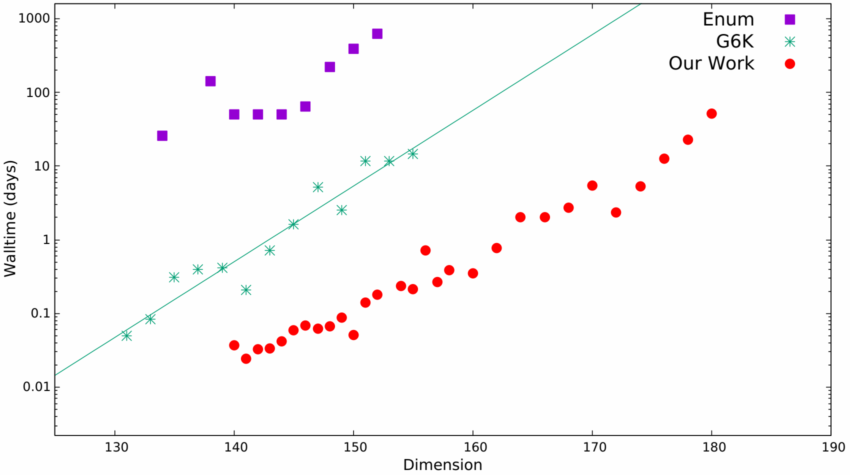
- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} x_i \boldsymbol{b}_i$:
  - $[x_1, \ldots, x_d]$. ($2d$)
  - $[\langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_1 \rangle / \left\|\tilde{\boldsymbol{b}}_1\right\|, \ldots, \langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_d \rangle / \left\|\tilde{\boldsymbol{b}}_d\right\|]$. ($4d$)
  - $\|\boldsymbol{y}\|^2$. (8)
  - Unique Identifier (8)
  - Lift target $\boldsymbol{t}$. ($4l$)
  - Dual Hash ($4l$)
  - Popcount ($d/4$)
- Remove all except $\boldsymbol{x}$, $\|\boldsymbol{y}\|^2$ and unique identifier.
- Reduces memory by $\pm$**60**%.

Saving Memory

- G6K stores lot's of information per vector $\boldsymbol{y} = \sum_{i=1}^{d} x_i \boldsymbol{b}_i$:
  - $[x_1, \ldots, x_d]$. $(\mathbf{2}d)$
  - $[\langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_1 \rangle / \left\| \tilde{\boldsymbol{b}}_1 \right\|, \ldots, \langle \boldsymbol{y}, \tilde{\boldsymbol{b}}_d \rangle / \left\| \tilde{\boldsymbol{b}}_d \right\|]$. $(\mathbf{4}d)$
  - $\|\boldsymbol{y}\|^2$. $(\mathbf{8})$
  - Unique Identifier $(\mathbf{8})$
  - Lift target $\boldsymbol{t}$. $(\mathbf{4}l)$
  - Dual Hash $(\mathbf{4}l)$
  - Popcount $(d/\mathbf{4})$
- Remove all except x, $\|\boldsymbol{y}\|^2$ and unique identifier.
- Reduces memory by $\pm\mathbf{60}\%$.
- Compute everything on the GPU, overhead of $O(B \cdot d^2)$ for a bucket size B.

New SVP records

Dimension 180!



maximum RAM size of 1.5TB reached for **180**.

Conclusion

- Lattice sieving algorithms can efficiently be implemented on GPUs.

Conclusion

- Lattice sieving algorithms can efficiently be implemented on GPUs.
- Memory bottlenecks disappear when buckets are large enough.

Conclusion

- Lattice sieving algorithms can efficiently be implemented on GPUs.
- Memory bottlenecks disappear when buckets are large enough.
- Extra benefit of saving memory with negligible overhead.

Conclusion

- Lattice sieving algorithms can efficiently be implemented on GPUs.
- Memory bottlenecks disappear when buckets are large enough.
- Extra benefit of saving memory with negligible overhead.
- BDGL beats BGJ1 in practice on CPUs, but the cross-over for GPUs lies much higher.

Bibliography

- [BGJ15] A. Becker,N. Gama, A. Joux. Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search.
- [BDGL16] A. Becker, L. Ducas, N. Gama, T. Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving.
- [Duc18] L. Ducas, Shortest vector from lattice sieving: A few dimensions for free.
- [LM18] T. Laarhoven, A. Mariano, Progressive lattice sieving.
- [G6K] M.R. Albrecht, L. Ducas, G. Herold, E. Kirshanova, E.W. Postlethwaite, and M. Stevens, 2019. The general sieve kernel and new records in lattice reduction.