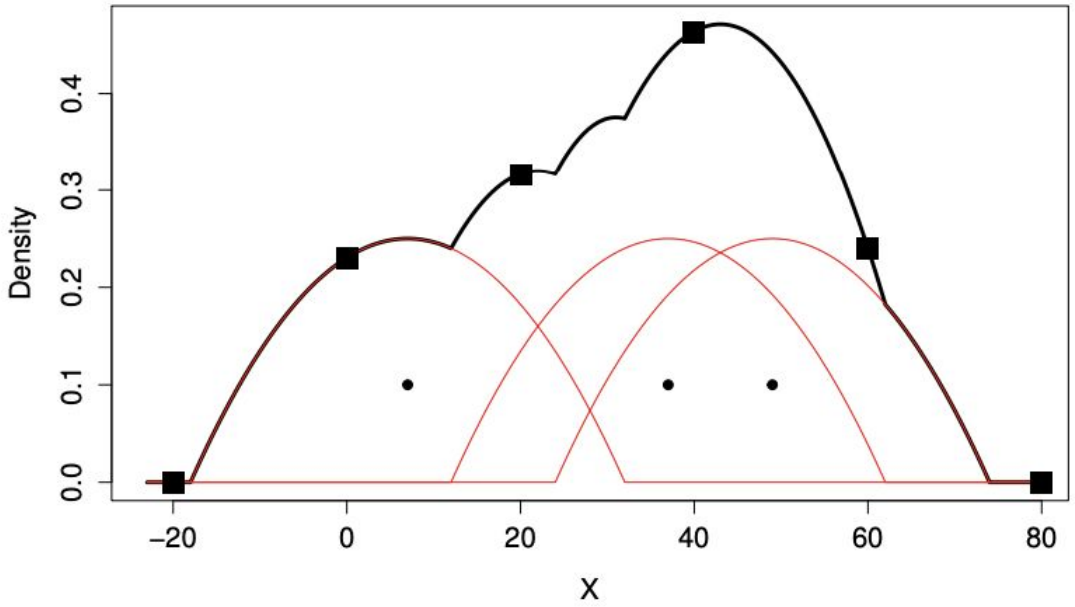


VU Visualisierung 2 (186.833)

Kernel density estimation in accelerators

Research Paper Implementation

Franciszek Szewczyk (12510815) / Sam Engel (12532780)

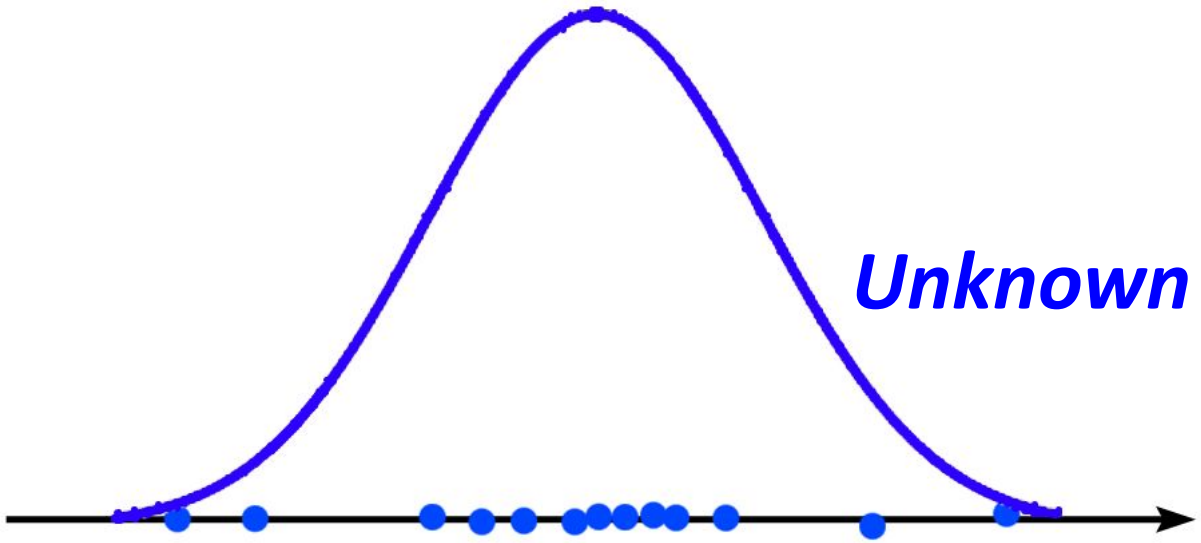
Institute of Visual Computing & Human-Centered Technology, TU Wien, Austria

Motivation

1. You observe a ***limited set of samples*** from some unknown distribution

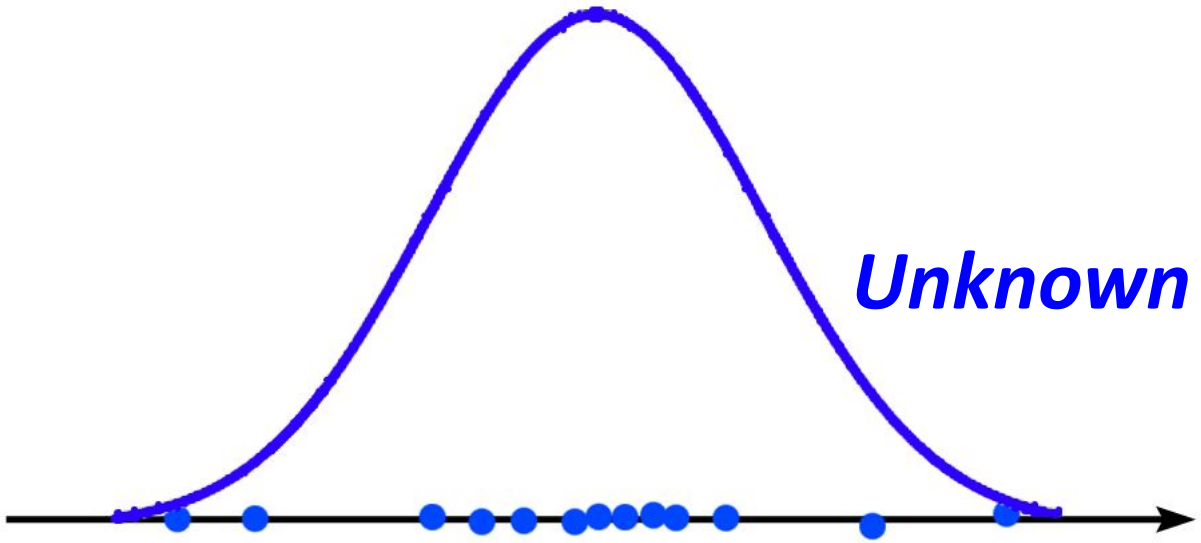
Motivation

1. You observe a ***limited set of samples*** from some unknown distribution
2. How do we ***estimate the original distribution*** it came from?



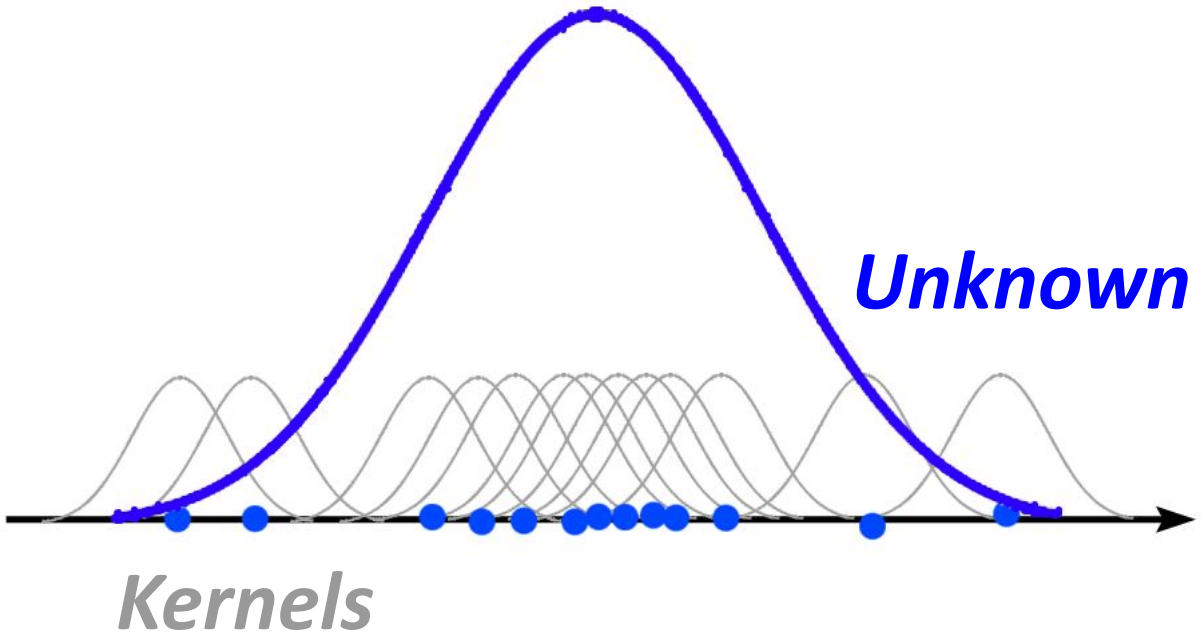
Motivation

1. You observe a ***limited set of samples*** from some unknown distribution
2. How do we ***estimate the original distribution*** it came from?
   a. If we knew it, we could simulate more samples
   b. Visualize the data
   c. Understand the problem at hand better



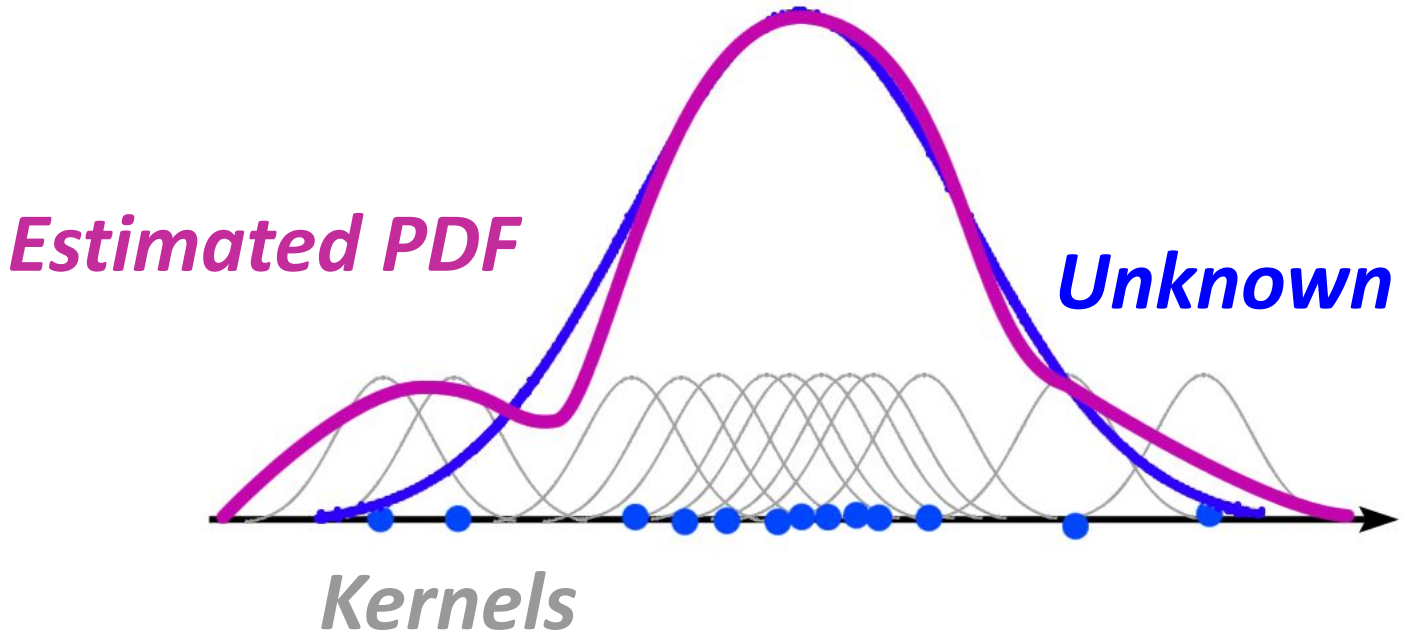
Motivation

1. You observe a ***limited set of samples*** from some unknown distribution
2. How do we ***estimate the original distribution*** it came from?
3. We can add a small ***kernel*** around each sample



Motivation

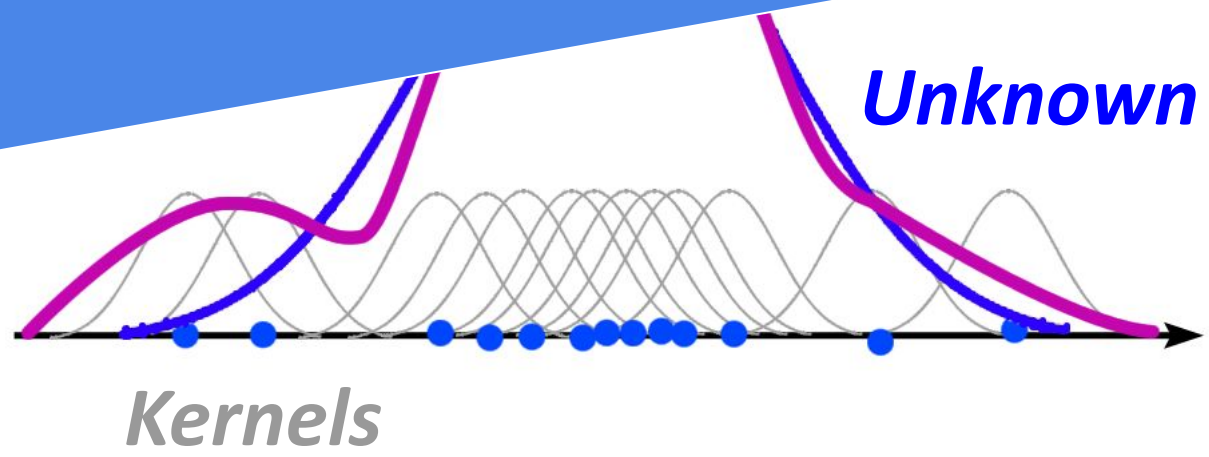
1. You observe a ***limited set of samples*** from some unknown distribution
2. How do we ***estimate the original distribution*** it came from?
3. We can add a small ***kernel*** around each sample
4. And ***evaluate their sums*** in a dense grid



Motivation

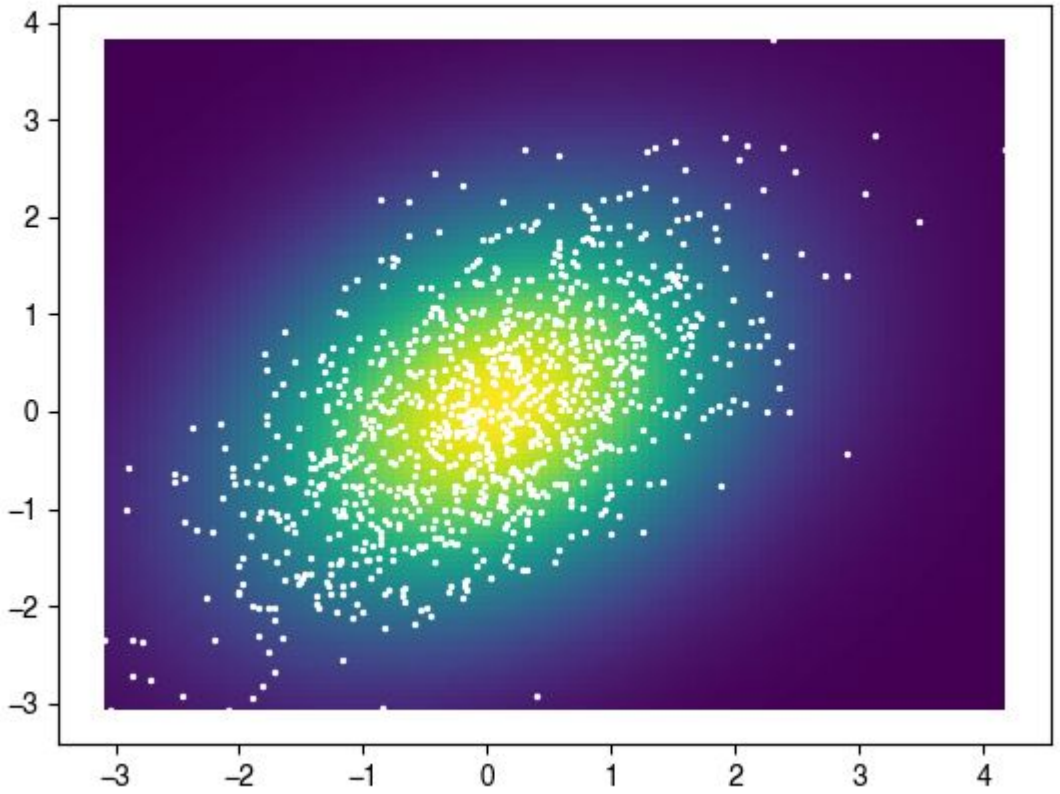
1. You observe a ***limited set of samples*** from some unknown distribution
2. How do we ***estimate the original distribution*** it came from?
3. We can add a small ***kernel*** around each sample
4. And ***evaluate their sums*** in a dense grid

Kernel Density Estimation!



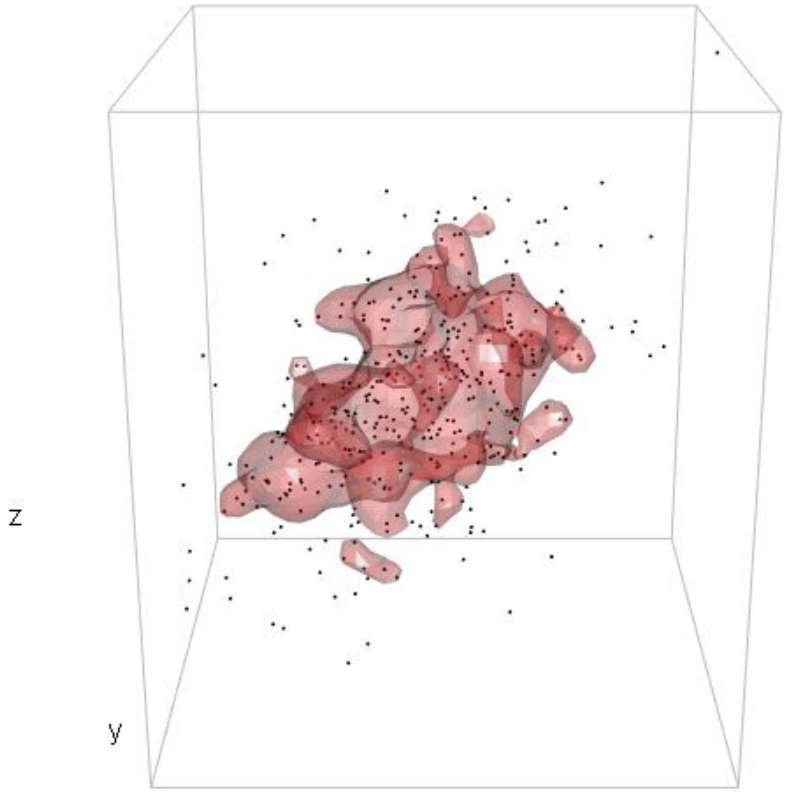
More Dimensions!

1. The basic principle is easily extended to **2D**



More Dimensions!

1. The basic principle is easily extended to **2D**
2. and **3D**



Slow…

1. Naive algorithm is **quite slow**:
   a. For every sample
      i. For every evaluated grid cell
         1. Accumulate the value of the kernel

Slow…

1. Naive algorithm is **quite slow**
2. But we can **parallelize**, across:
   a. Each sample
   b. Each grid cell a sample touches

Slow…

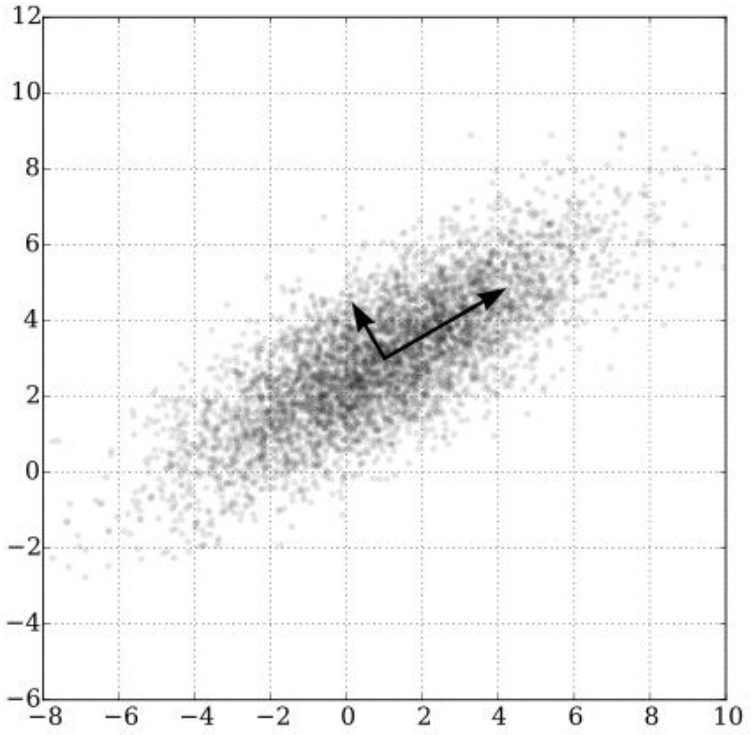
1. Naive algorithm is **quite slow**
2. But we can **parallelize**, across:
   a. Each sample
   b. Each grid cell a sample touches
3. And we can precompute exactly which cells will be affected*

$$\left(\quad \right)^2$$

**Lopez-Novoa et al. 2016*

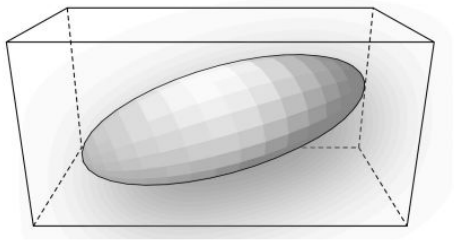
How to figure out the 3D ellipsoid kernel?

1. Covariance matrix of the data encodes the “ellipsoid” shape of the data
2. Ellipsoid Matrix with the shape of the data is the ***inverse of the covariance***



Chop

1. For each sample, calculate its 3D bounding box



Chop & Crop

1. For each sample, calculate its 3D bounding box
2. ***Chop the ellipsoid kernel*** along the Z axis

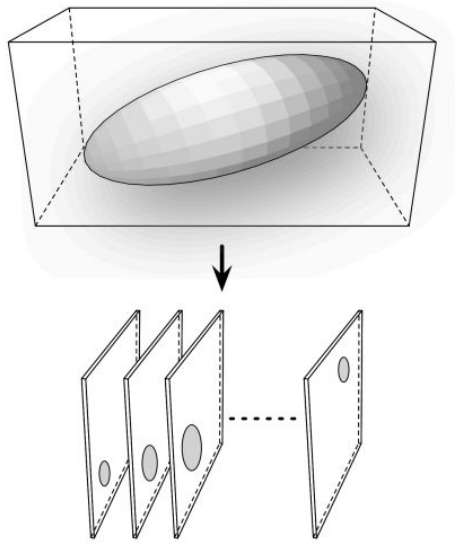


Fig. 2 Chopping a 3D bounding box into 2D slices

Chop & Crop

1. For each sample, calculate its 3D bounding box
2. ***Chop the ellipsoid kernel*** along the Z axis
3. ***Crop each 2D slice*** so that only the cells inside the kernel are considered

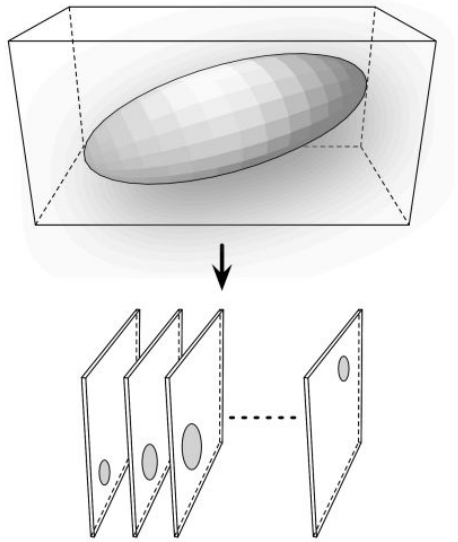


Fig. 2 Chopping a 3D bounding box into 2D slices

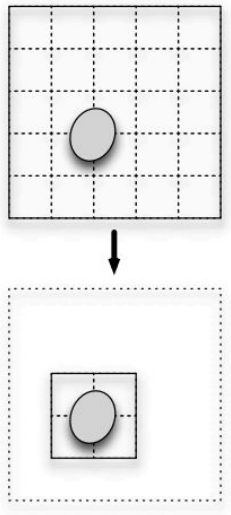


Fig. 3 Cropping a 2D slice to obtain a minimum-size bounding rectangle

Chop & Crop

1. For each sample, calculate its 3D bounding box
2. ***Chop the ellipsoid kernel*** along the Z axis
3. ***Crop each 2D slice*** so that only the cells inside the kernel are considered
4. Launch a parallel thread ***for each cell of each cropped slice for each sample***

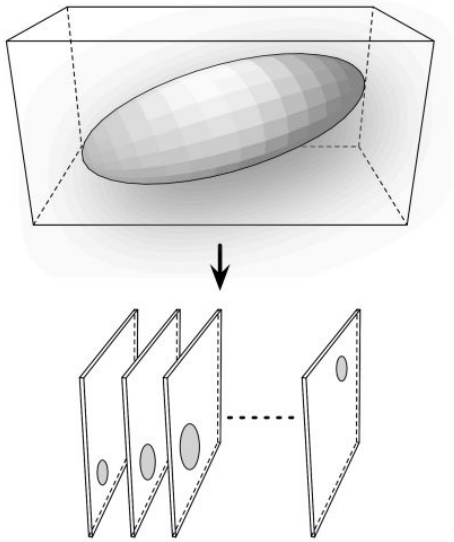


Fig. 2 Chopping a 3D bounding box into 2D slices

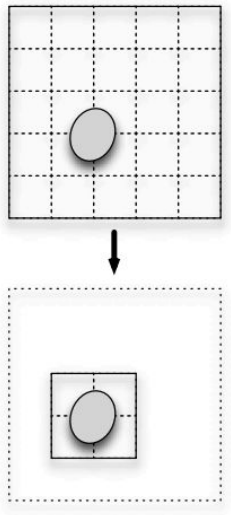
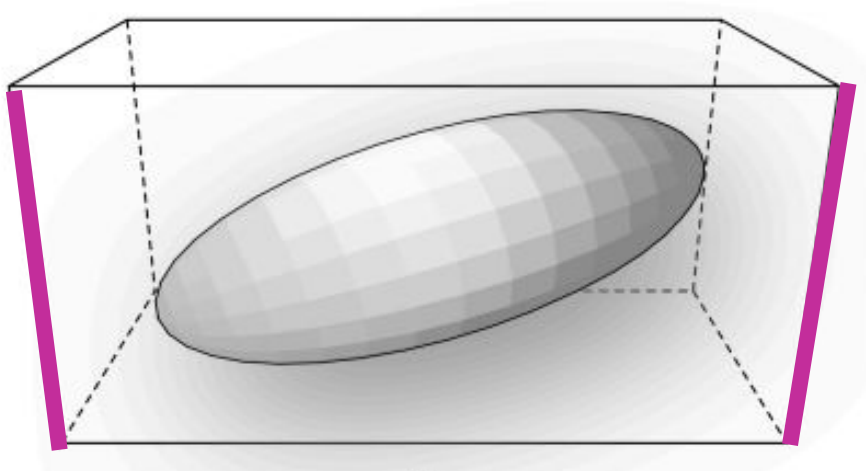


Fig. 3 Cropping a 2D slice to obtain a minimum-size bounding rectangle

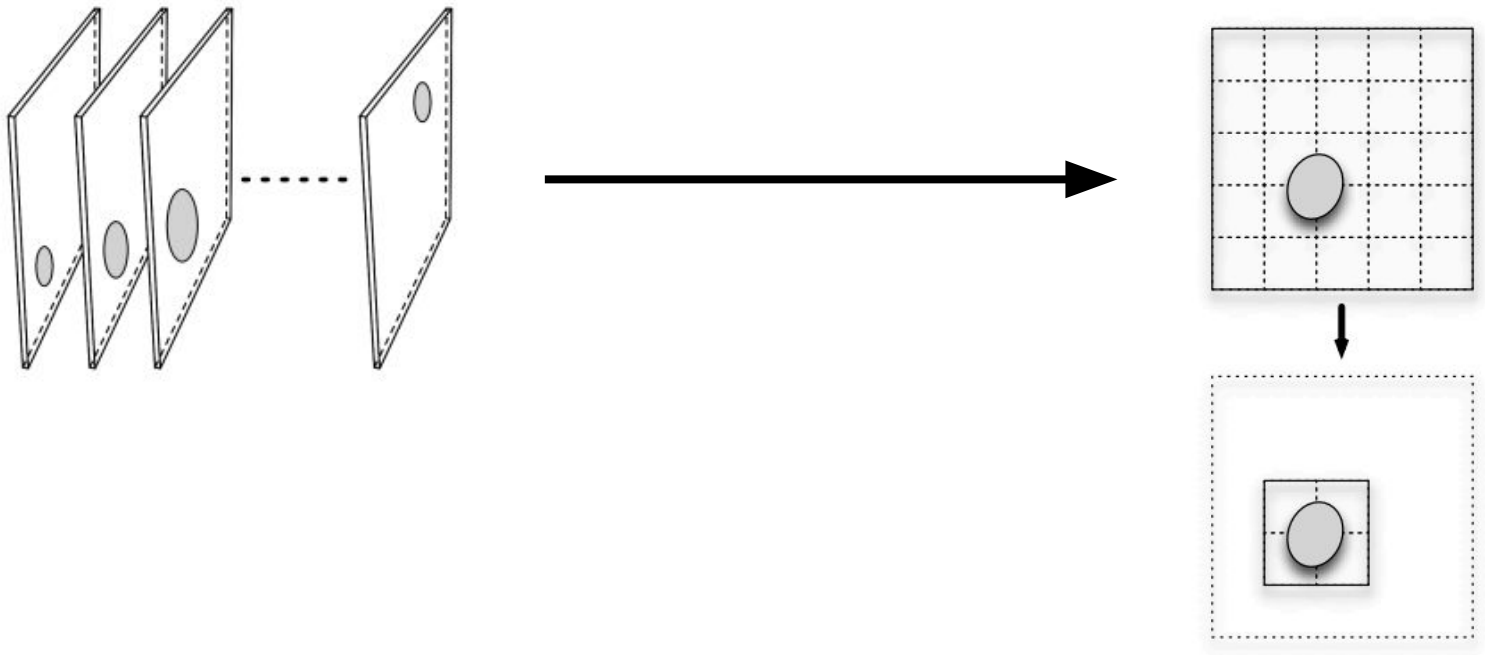
Implementation - ***Chop Kernel***

1. CUDA ***thread considers a single sample***
2. Stores the ***ellipsoid kernel boundaries*** along the ***Z*** dimension
3. No race conditions



Implementation - ***Crop Kernel***

1. CUDA ***thread considers a single slice of a single sample***
2. Stores the ***X and Y boundaries*** within that slice
3. No race conditions



Implementation - ***Crop Kernel***

1. CUDA ***thread considers a single slice of a single sample***
2. Stores the ***valid Y boundaries*** within that slice
3. No race conditions

Mathematical Interlude

Implementation - *Crop Kernel*

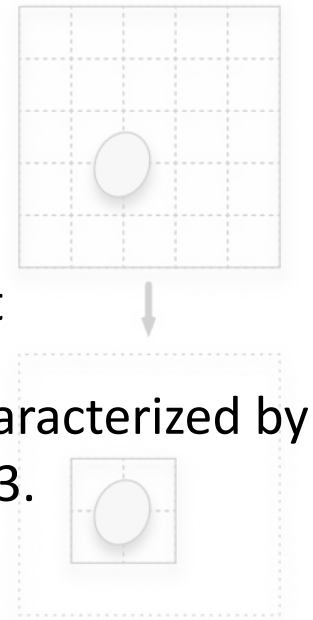
1. CUDA *thread considers a single slice of a single sample*
2. Stores the *X and Y boundaries* within that slice
3. No race conditions

Mathematical Interlude

The general form of a rotated, translated ellipse is a quadratic equation: $Ax^2 + Bxy + Cy^2 + Dx + Ey + F < 1$

This can be represented with matrices:

```
[x y 1] * [  A    B/2   D/2 ] * [x]
          [ B/2    C    E/2 ]   [y]
          [ D/2   E/2    F  ]   [1]
```



…or $x^T A x < 1$ for short

This all generalizes to higher dimensions. Our ellipsoid kernel is characterized by a 4x4 matrix, except that because it has no translation, we can get away with 3x3.

Implementation - ***Crop Kernel***

1. CUDA ***thread considers a single slice of a single sample***
2. Stores the ***X and Y boundaries*** within that slice
3. No race conditions

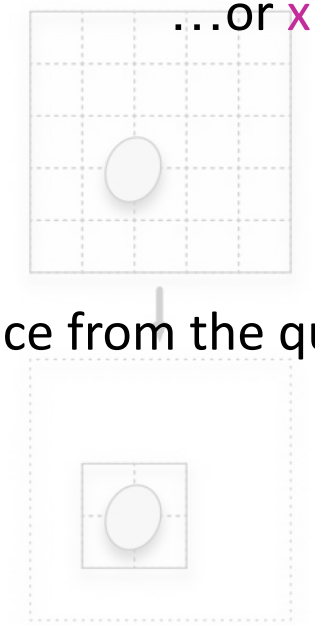
Mathematical Interlude

The general form of a rotated, translated ellipse is a quadratic equation: $Ax^2 + Bxy + Cy^2 + Dx + Ey + F < 1$

…or $x^T A x < 1$ for short

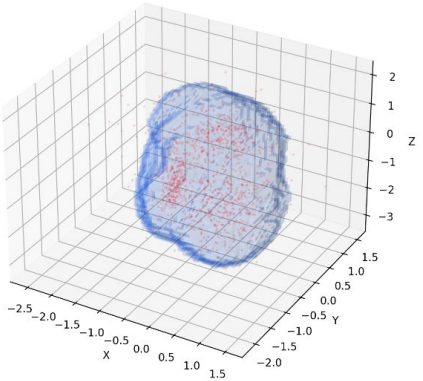
To get a 2D slice of a 3D ellipsoid:

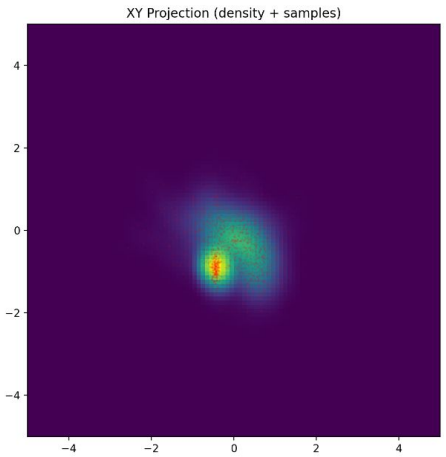
1. ***Substitute the desired z value*** into the 3D quadratic equation
2. Now you have a ***2D quadratic equation!***
3. ***Reconstruct*** the semi-axes and translation of the 2D elliptic slice from the quadratic coefficients

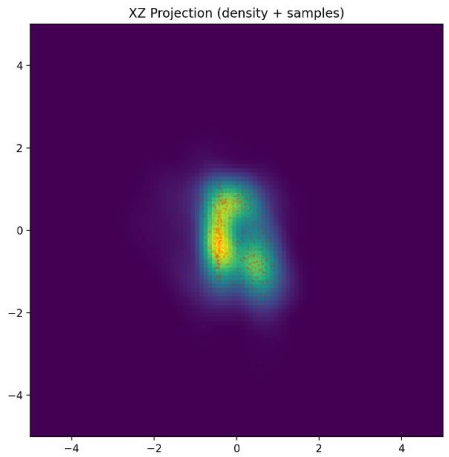


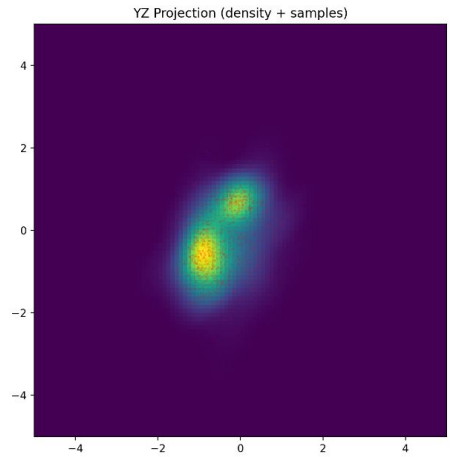
Implementation - ***KDE Kernel***

1. CUDA ***thread considers a single cell affected by a single sample***
2. Checks the value of the ellipsoid kernel
3. And ***atomically accumulates the result*** in the final grid









Implementation - *Visualization*

1. In the original paper, densities were accumulated on the CPU, which was a serious ***bottleneck***

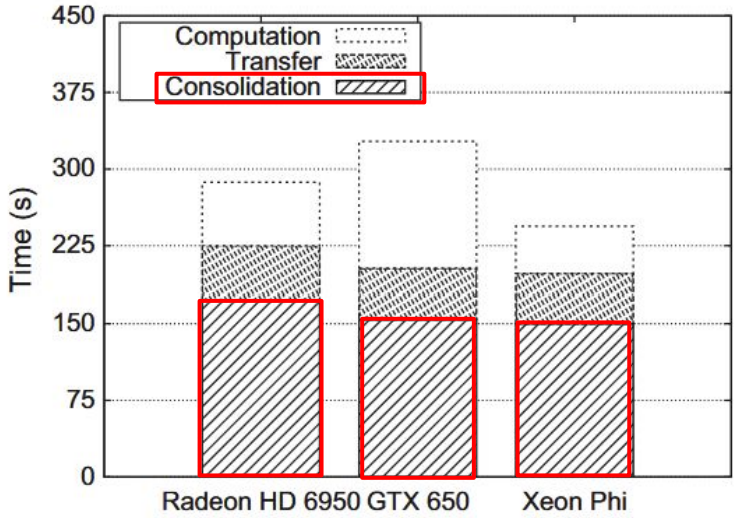


Fig. 8 Dissected execution time of OpenCL S-KDE. 1M dataset and 194M evaluation points

Implementation - *Visualization*

1. In the original paper, densities were accumulated on the CPU, which was a serious ***bottleneck***
2. By doing everything on the GPU, ***we can visualize the results directly***

Implementation - *Visualization*

1. In the original paper, densities were accumulated on the CPU, which was a serious ***bottleneck***
2. By doing everything on the GPU, ***we can visualize the results directly***
3. We can also visualize ***the algorithm itself!***

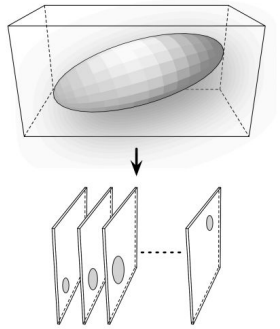


Fig. 2 Chopping a 3D bounding box into 2D slices

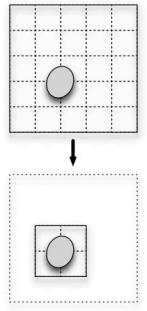


Fig. 3 Cropping a 2D slice to obtain a minimum-size bounding rectangle

Implementation - *Visualization*

1. In the original paper, densities were accumulated on the CPU, which was a serious ***bottleneck***
2. By doing everything on the GPU, ***we can visualize the results directly***
3. We can also visualize ***the algorithm itself!***

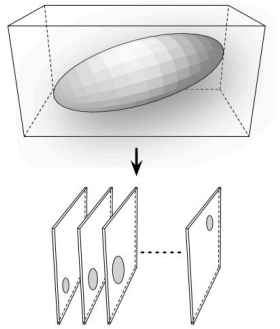


Fig. 2 Chopping a 3D bounding box into 2D slices

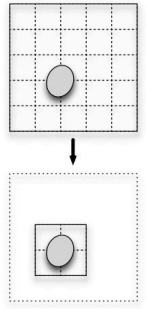
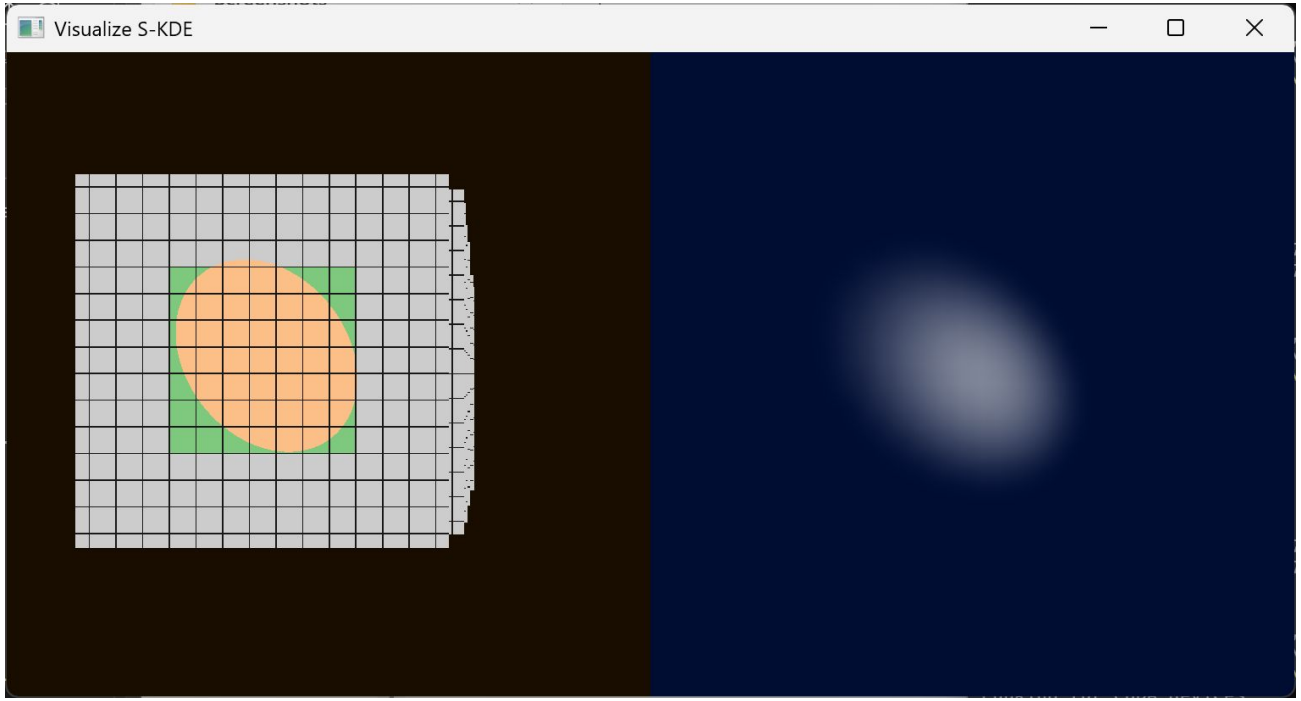


Fig. 3 Cropping a 2D slice to obtain a minimum-size bounding rectangle



Implementation - *Visualization*

1. In the original paper, densities were accumulated on the CPU, which was a serious ***bottleneck***
2. By doing everything on the GPU, ***we can visualize the results directly***
3. We can also visualize ***the algorithm itself!***

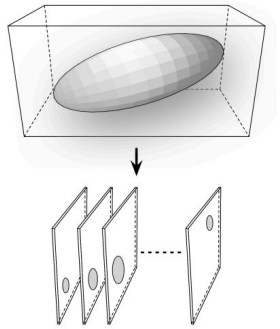


Fig. 2 Chopping a 3D bounding box into 2D slices

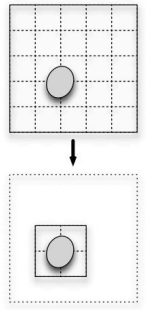
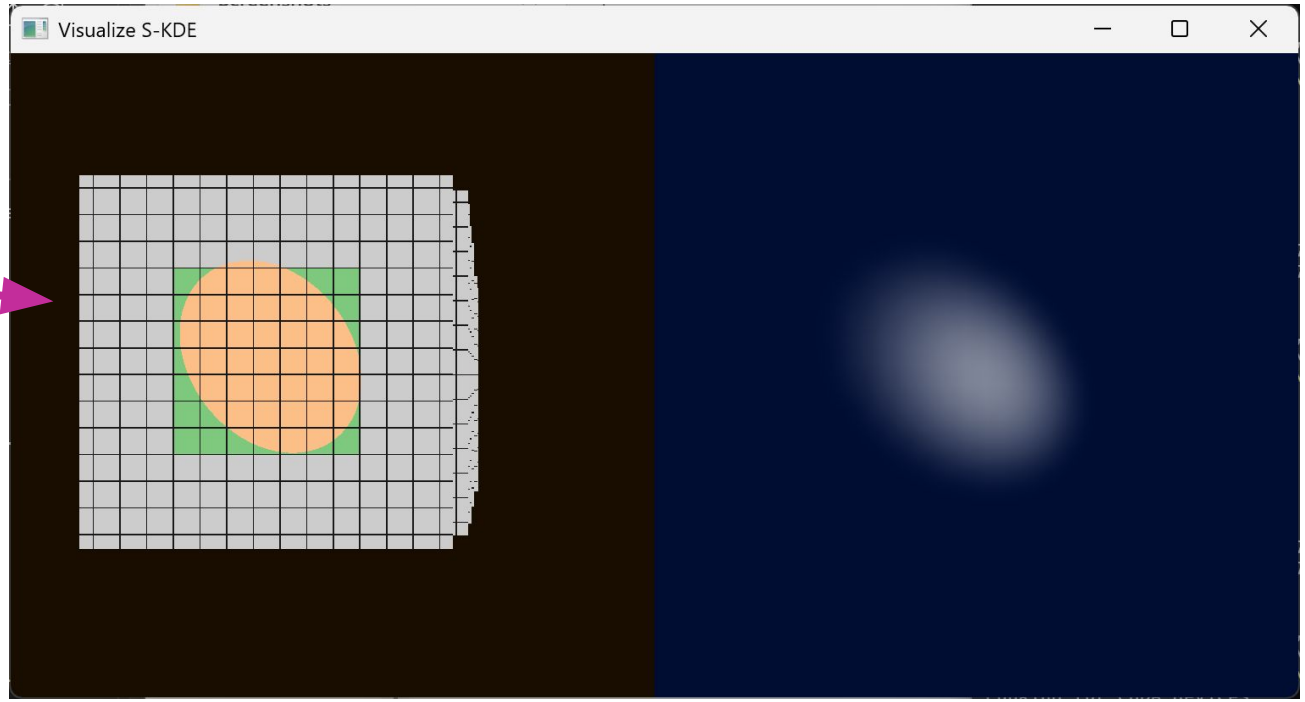


Fig. 3 Cropping a 2D slice to obtain a minimum-size bounding rectangle



Implementation - *Visualization*

1. In the original paper, densities were accumulated on the CPU, which was a serious ***bottleneck***
2. By doing everything on the GPU, ***we can visualize the results directly***
3. We can also visualize ***the algorithm itself!***

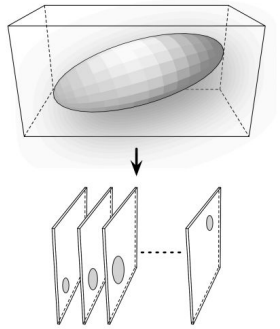


Fig. 2 Chopping a 3D bounding box into 2D slices

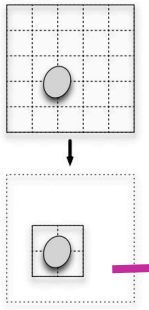
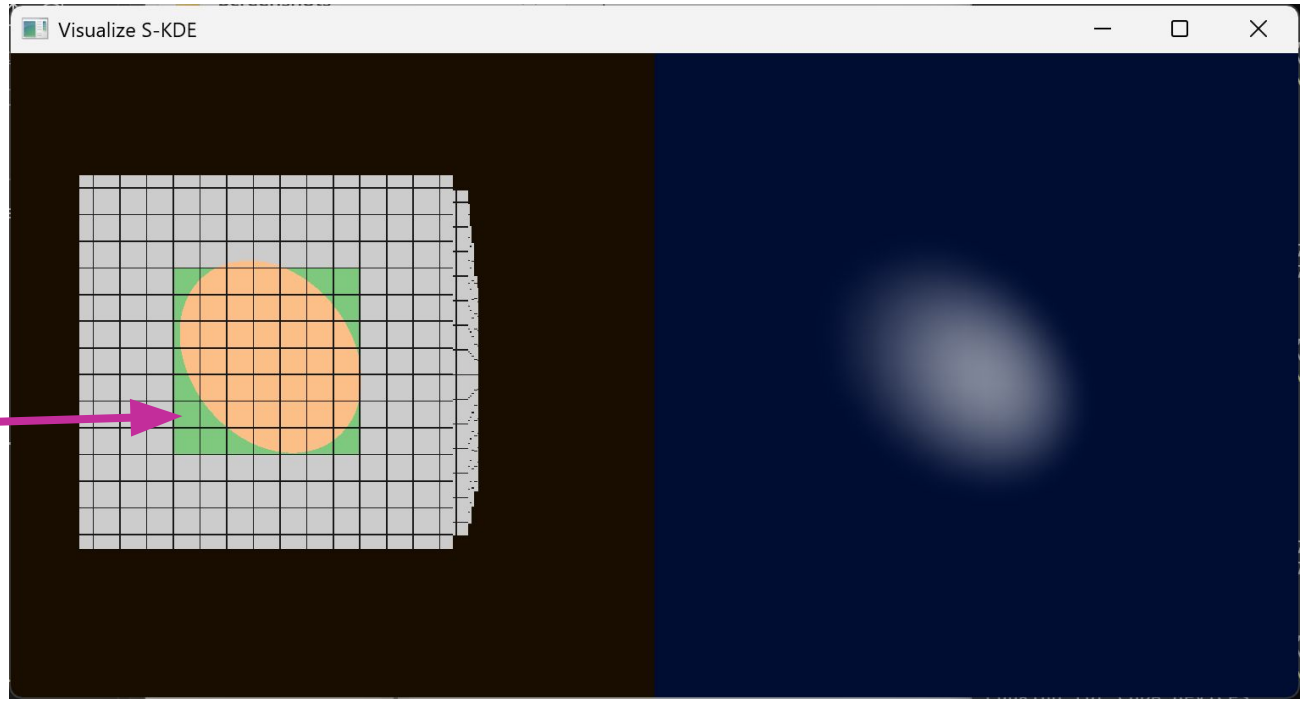


Fig. 3 Cropping a 2D slice to obtain a minimum-size bounding rectangle



Implementation - *Visualization*

1. In the original paper, densities were accumulated on the CPU, which was a serious ***bottleneck***
2. By doing everything on the GPU, ***we can visualize the results directly***
3. We can also visualize ***the algorithm itself!***

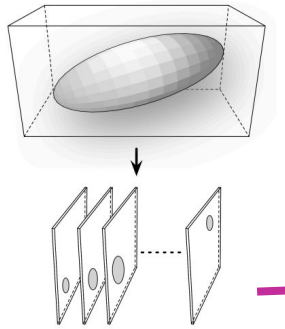


Fig. 2 Chopping a 3D bounding box into 2D slices

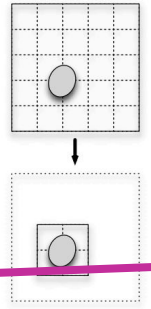
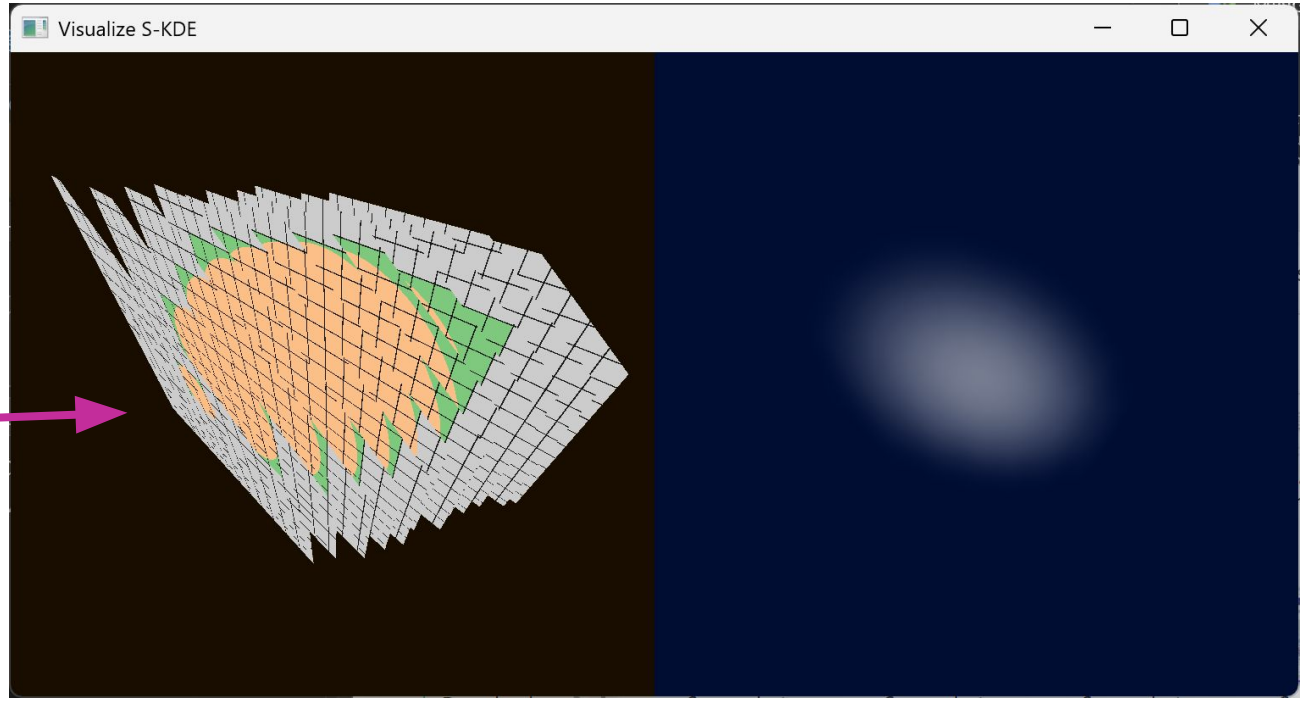


Fig. 3 Cropping a 2D slice to obtain a minimum-size bounding rectangle



Implementation - *Visualization*

1. In the original paper, densities were accumulated on the CPU, which was a serious ***bottleneck***
2. By doing everything on the GPU, ***we can visualize the results directly***
3. We can also visualize ***the algorithm itself!***

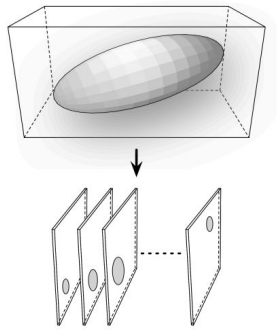


Fig. 2 Chopping a 3D bounding box into 2D slices

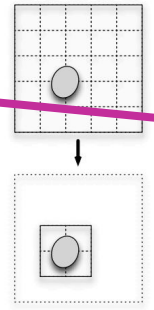
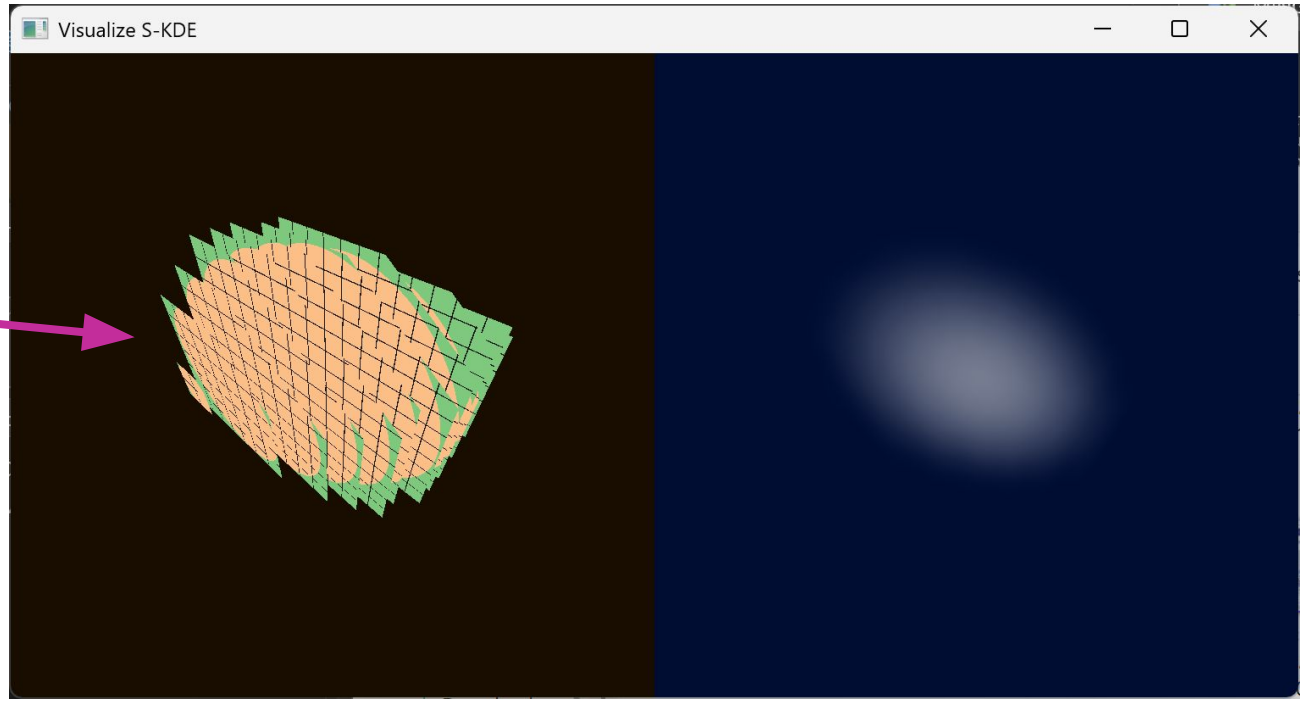
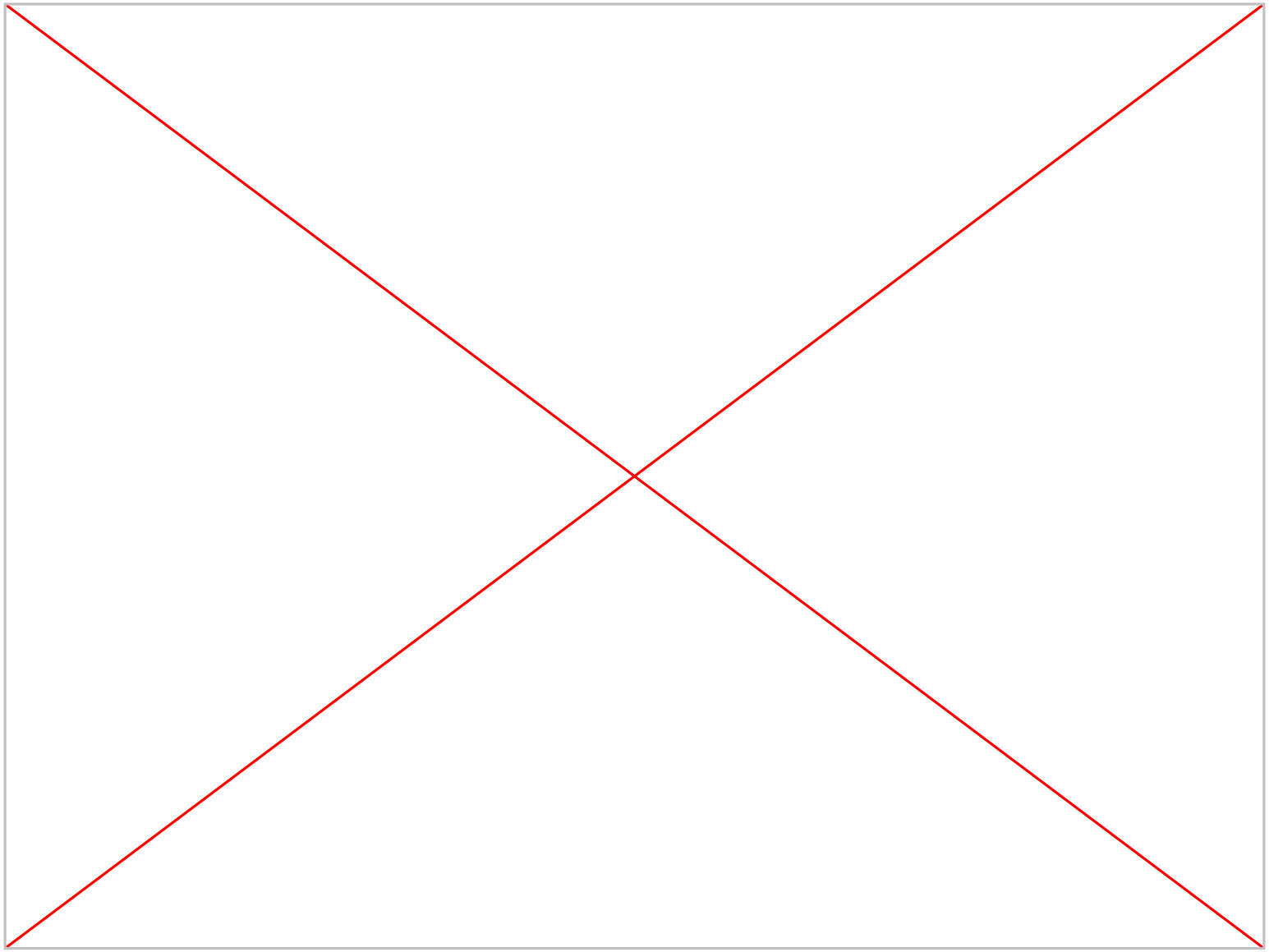


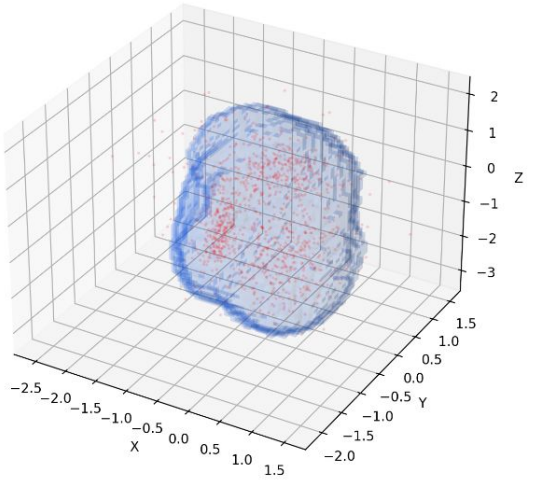
Fig. 3 Cropping a 2D slice to obtain a minimum-size bounding rectangle

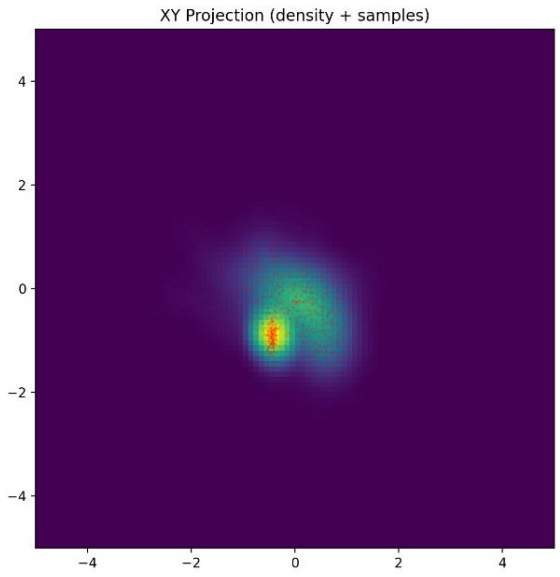


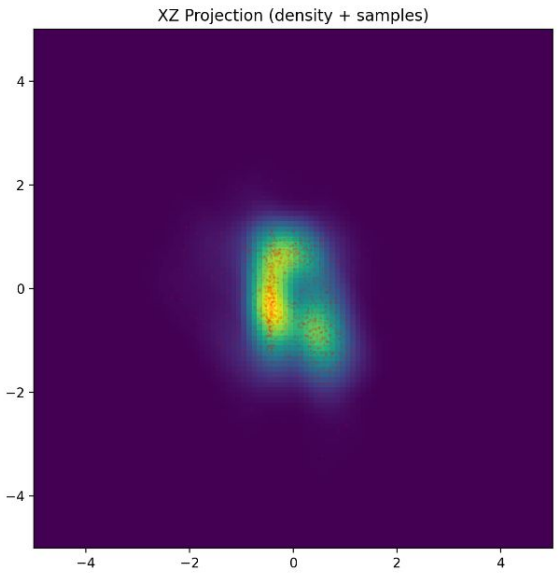
Implementation - ***Visualization***

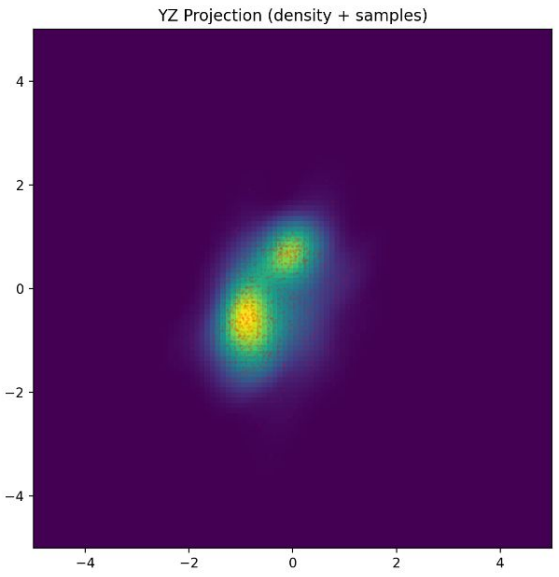


Example Result





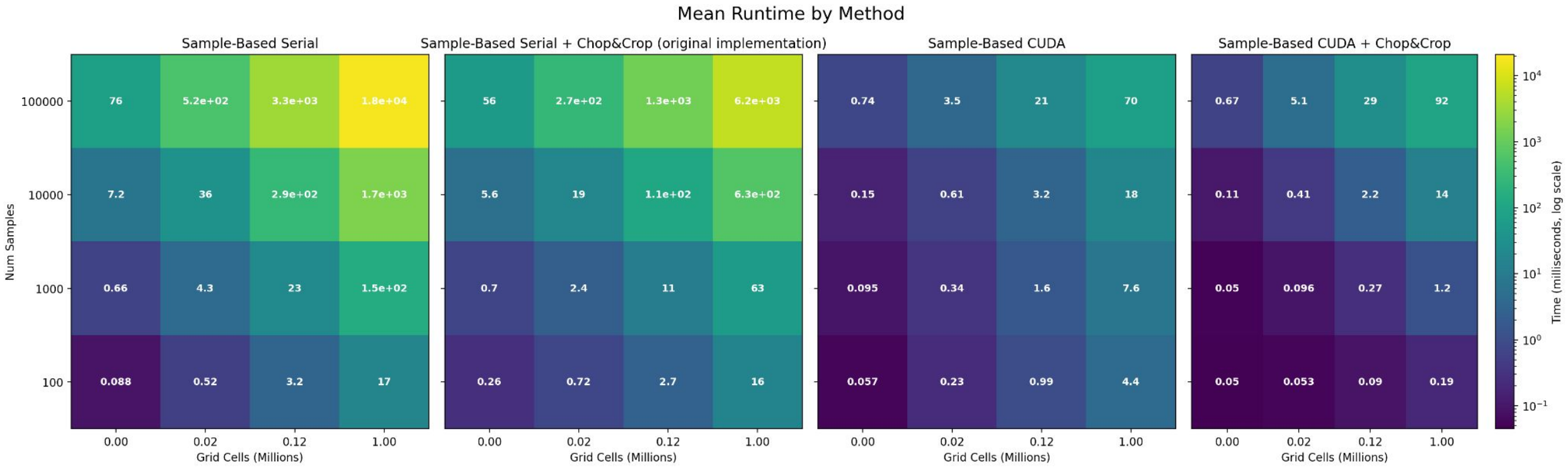




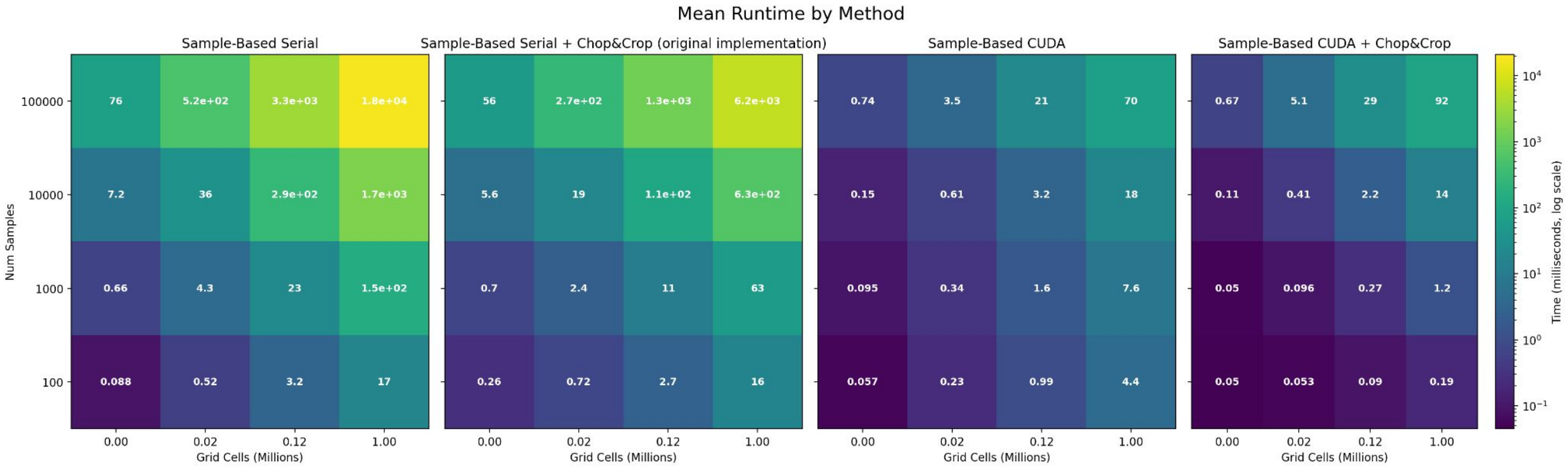
Experiments and Profiles

1. Compare timing against:
   a. CPU, serial
   b. CPU, serial + Chop&Crop (Original Reference Impl.)
   c. CUDA, parallel sample-wise
   d. CUDA, parallel sample-and-cell-wise + Chop&Crop
2. Generate samples from a ***set of 3D Gaussians***
3. 3 repetitions per testing condition
4. 3 warmup runs for CUDA

Experiments and Profiles



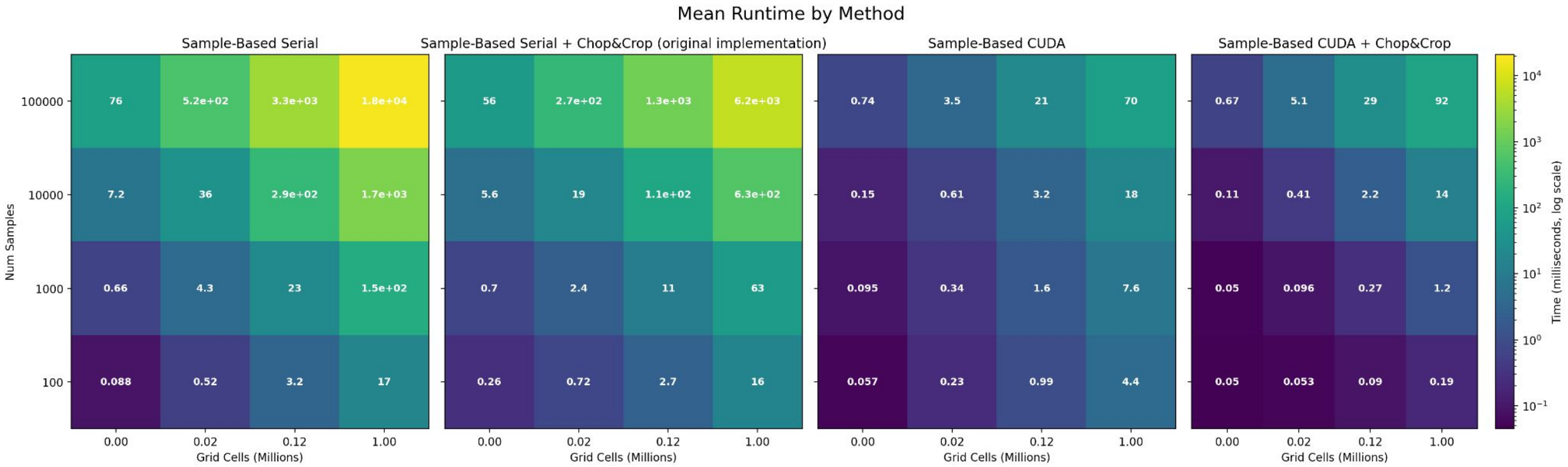
Experiments and Profiles



The more the cells, the slower we get

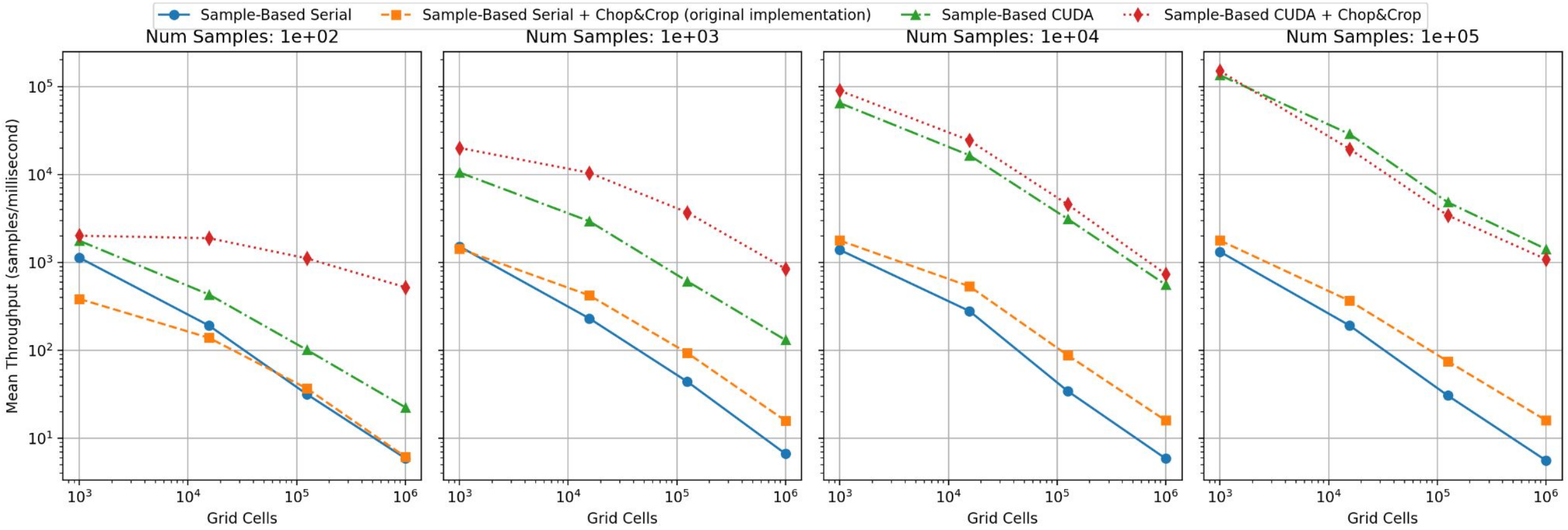
The more the samples, the slower we get

Experiments and Profiles

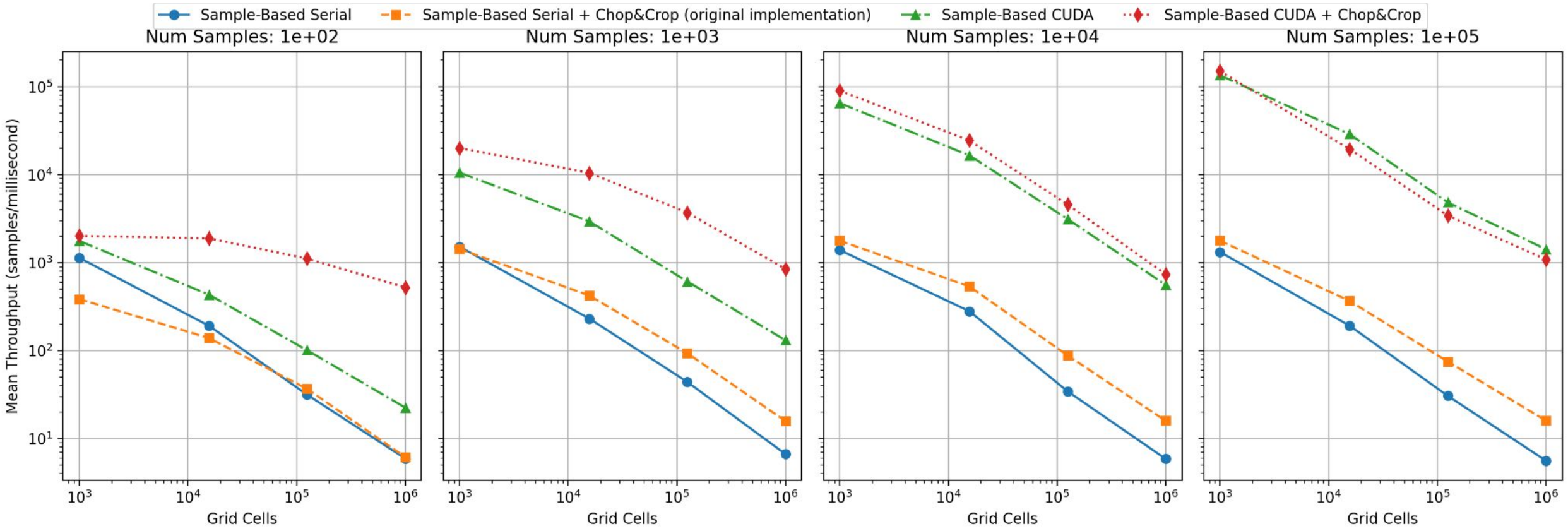


Both Chop&Crop and parallelism result in speedups

Experiments and Profiles

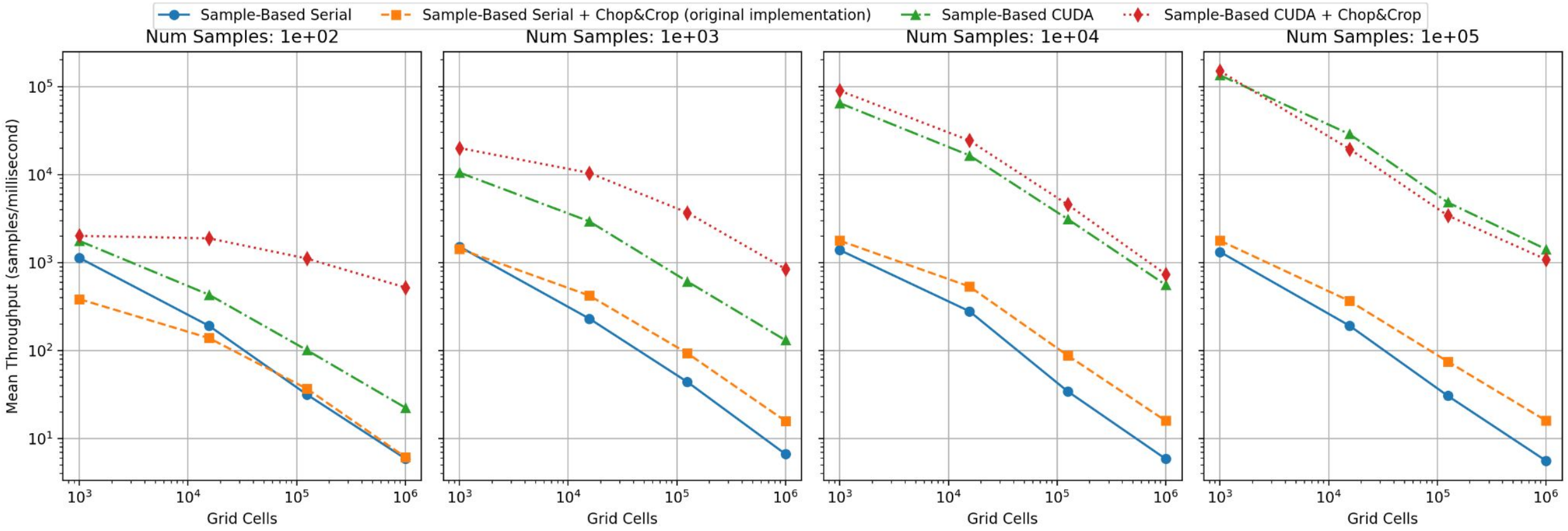


Experiments and Profiles



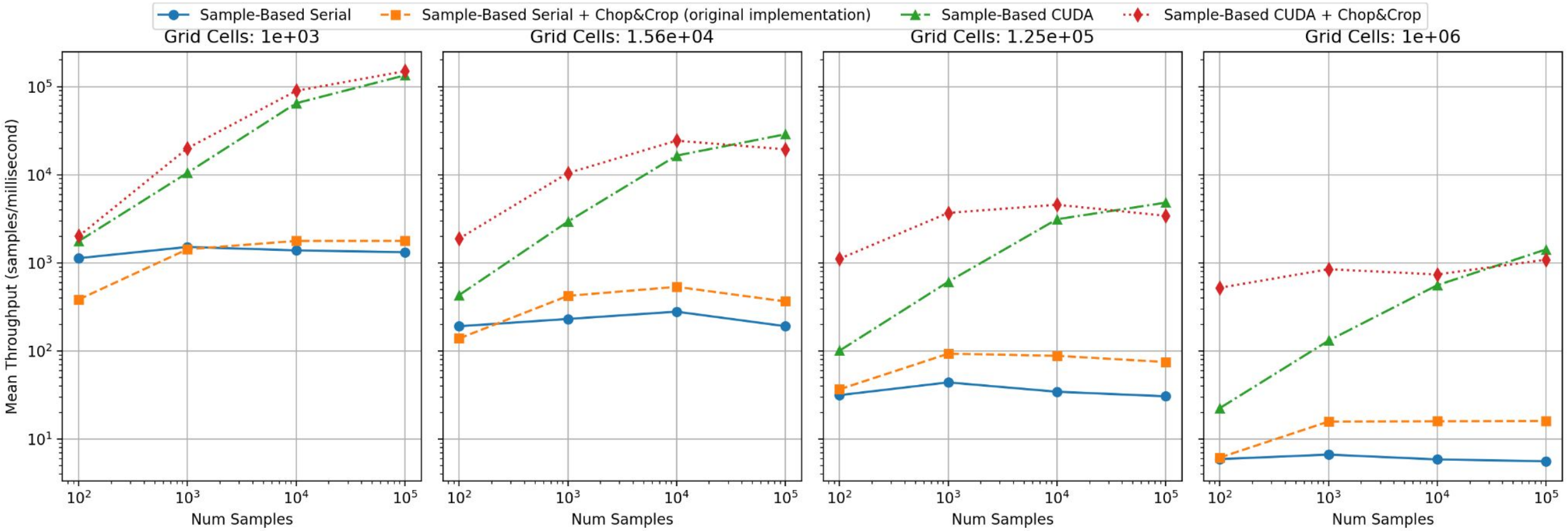
Chop&Crop improvements decay for large number of samples

Experiments and Profiles

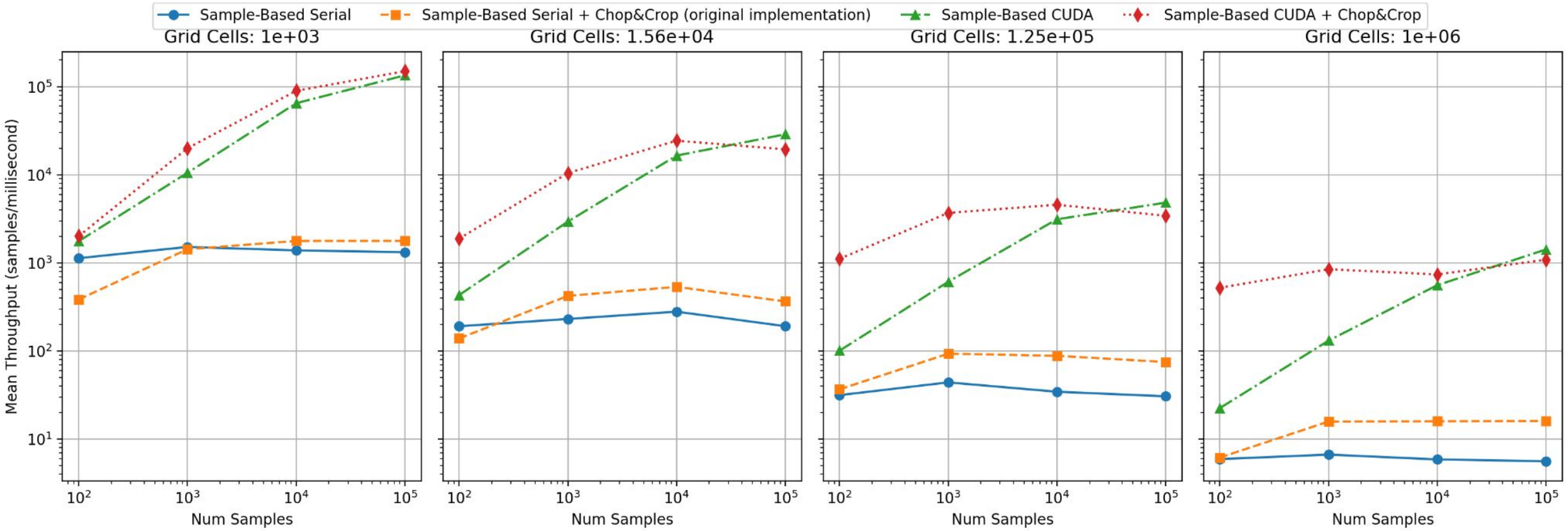


All methods suffer similar slowdown with the number of grid cells

Experiments and Profiles

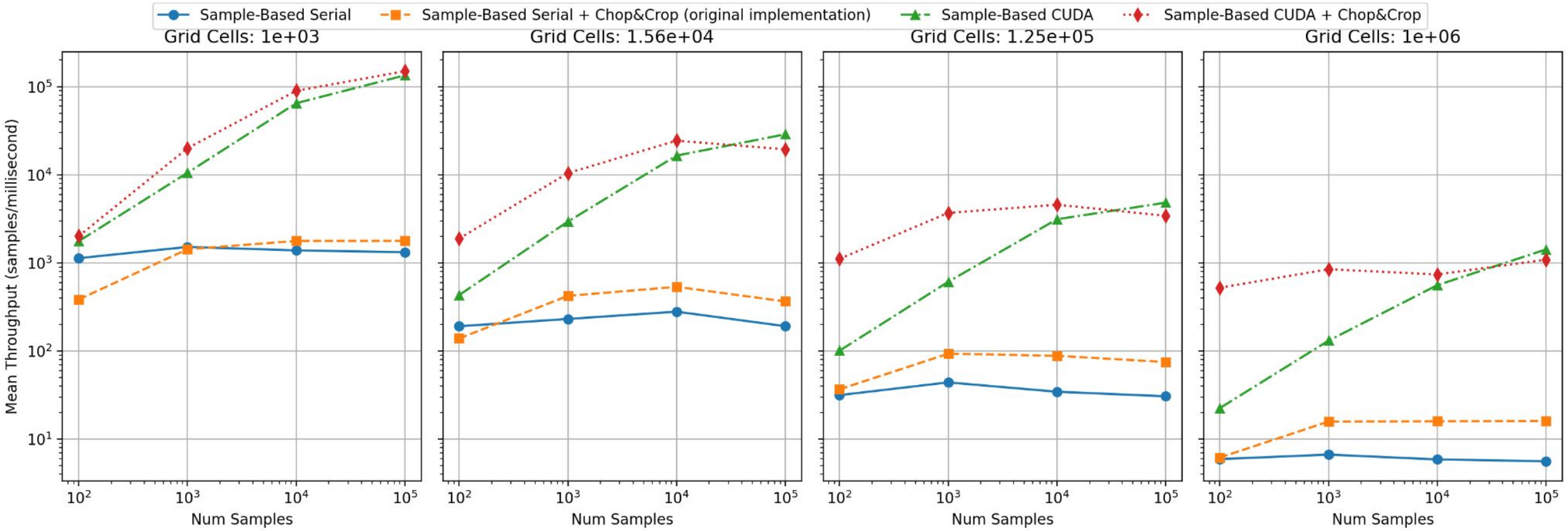


Experiments and Profiles



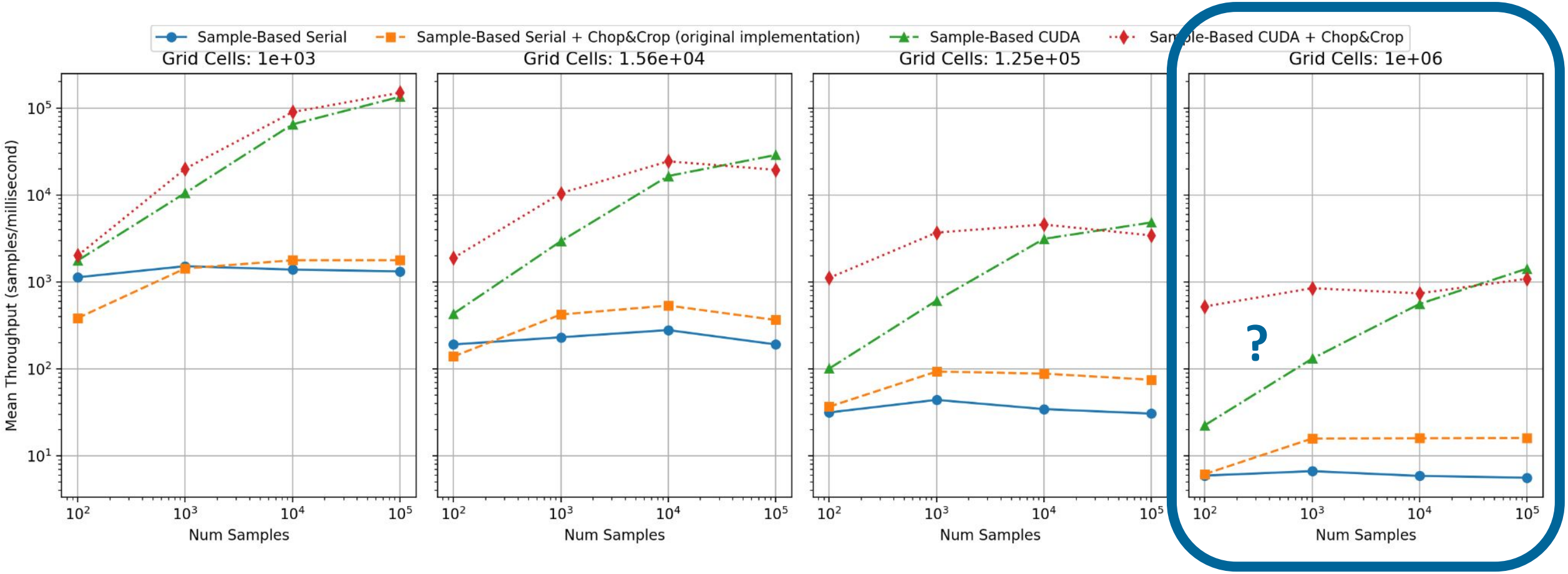
Serial Methods maintain its speed regardless of the dataset size

Experiments and Profiles

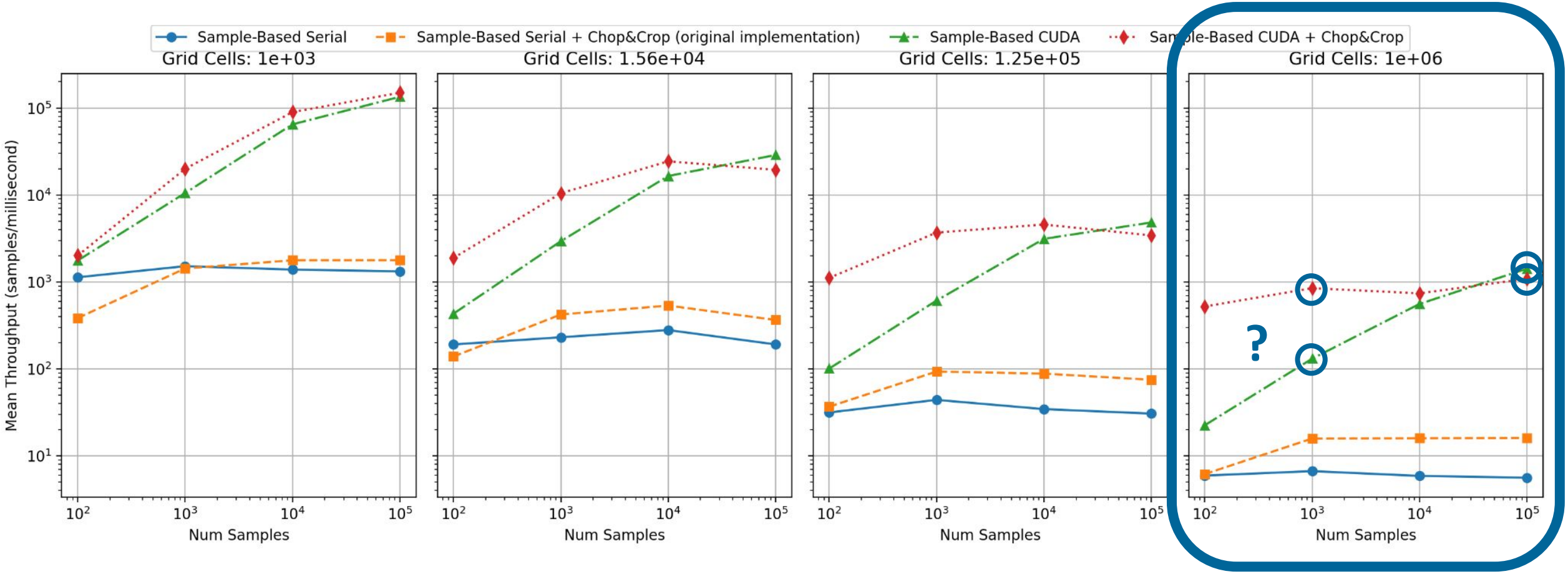


Parallel methods shine when given lots of samples

Experiments and Profiles



Experiments and Profiles



Experiments and Profiles

Sample-wise CUDA + Chop&Crop

Function Name	Demangled Name	Duration (1.01009e+09)
chopKernel	..chopKernel..Slice...	0.02
cropKernel	..cropKernel..Ellipse...	1.87
writeDensitiesKernel	..writeDensitiesKer...	251.62

Experiments and Profiles

L2 Cache Hit Rate (%)

	Naive CUDA	**CUDA SKDE** *(Final Kernel)*
1K samples	82.40	97.91
1M samples	98.89	98.67

Experiments and Profiles

Slowest Code Lines

	Naive CUDA	**CUDA SKDE** *(Final Kernel)*
1K samples	98% Location sm_60_atomic_functions.hpp:79 (0x500c5a6b0 in kde_kernel) sm_60_atomic_functions.hpp:79 (0x500c5a3f0 in kde_kernel) sm_60_atomic_functions.hpp:79 (0x500c5a970 in kde_kernel) sm_60_atomic_functions.hpp:79 (0x500c5aab0 in kde_kernel)	~100% Location cuda_skde.cuh:169 (0x500c61bc0 in writeDensitiesKernel)
1M samples	96% sm_60_atomic_functions.hpp:79 (0x500c5a3f0 in kde_kernel) sm_60_atomic_functions.hpp:79 (0x500c5a6b0 in kde_kernel) sm_60_atomic_functions.hpp:79 (0x500c5a970 in kde_kernel) sm_60_atomic_functions.hpp:79 (0x500c5aab0 in kde_kernel)	~100% Location cuda_skde.cuh:169 (0x500c61bc0 in writeDensitiesKernel)

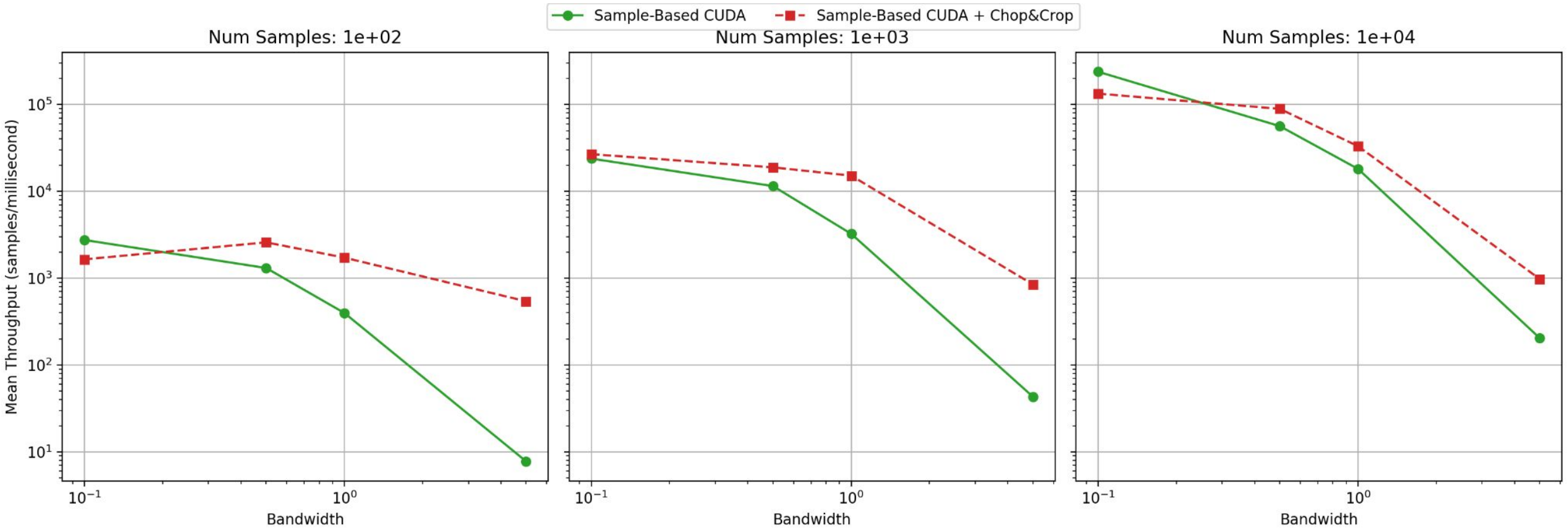
Experiments and Profiles

Slowest Code Lines

	Naive CUDA	CUDA SKDE *(Final Kernel)*
1K samples	98% sm_60_atomic_functions.hpp:79 (0x500c5a970 in kde_kernel) sm_60_atomic_functions.hpp:79 (0x500c5aab0 in kde_kernel)	Location cuda_skde.cuh:169 (0x500c61bc0 in writeDensitiesKernel)
1M samples	96% sm_60_atomic_functions.hpp:79 (0x500c5a3f0 in kde_kernel) sm_60_atomic_functions.hpp:79 (0x500c5a6b0 in kde_kernel) sm_60_atomic_functions.hpp:79 (0x500c5a970 in kde_kernel) sm_60_atomic_functions.hpp:79 (0x500c5aab0 in kde_kernel)	~100% Location cuda_skde.cuh:169 (0x500c61bc0 in writeDensitiesKernel)

```
atomicAdd(&pde[evaluationId], density);
```

Experiments and Profiles



Experiments and Profiles

Compute Throughput (%)

	Naive CUDA	**CUDA SKDE** *(Final Kernel)*
1K samples	9.58	85.39
1M samples	85.04	85.98

Experiments and Profiles

Achieved Occupancy (Theoretical) (%)

	Naive CUDA	**CUDA SKDE** *(Final Kernel)*
1K samples	16.59 (67)	62.56 (100)
1M samples	59.02 (67)	66.84 (100)

Experiments and Profiles

Achieved Occupancy (Theoretical) (%)

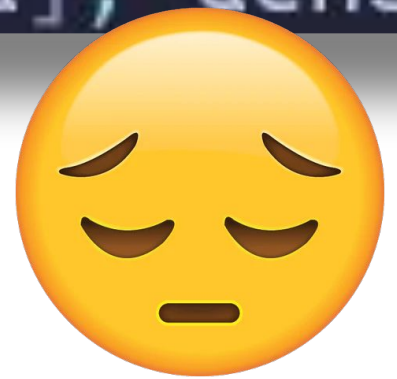
	Naive CUDA	**CUDA SKDE** *(Final Kernel)*
1K samples	16.59 (67)	62.56 (100)
1M samples	59.02 (67)	66.84 (100)

Difference likely due to thread-per-sample vs thread-per-sample-cell

Future Improvements

1. Faster, parallel, smarter density accumulation

```
atomicAdd(&pde[evaluationId], density);
```



Future Improvements

1. Faster, parallel, smarter density accumulation
2. Improved, coalesced data access for both samples and evaluation points

Thank you from Sam and Frank

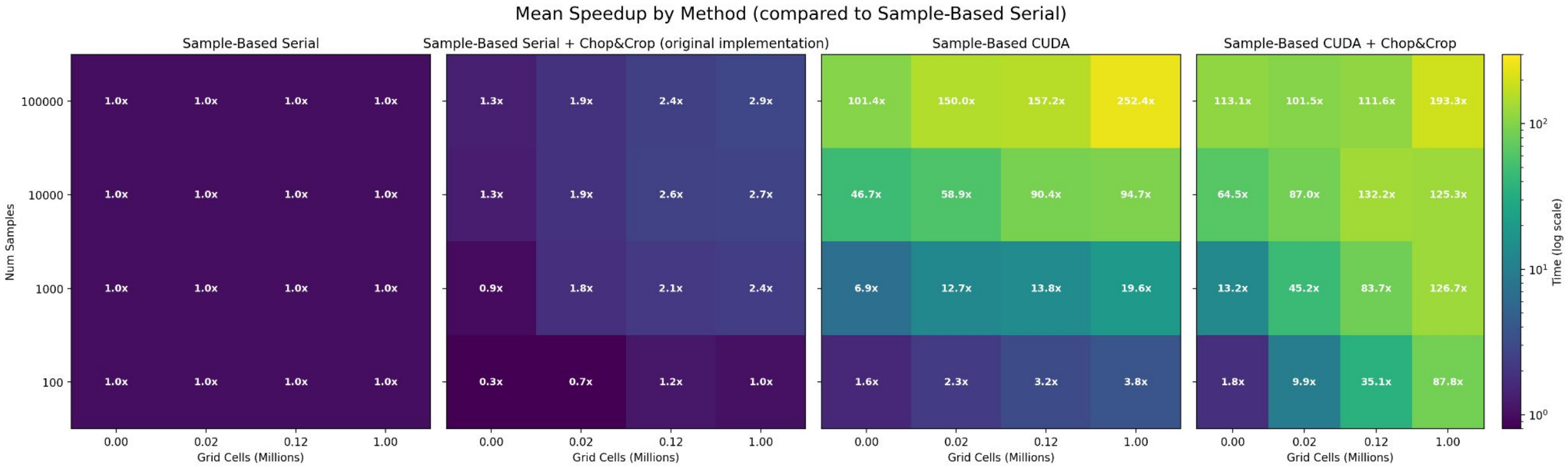
Lopez-Novoa, Unai and Mendiburu, Alexander and Miguel-Alonso, Jose

Kernel density estimation in accelerators: Implementation and performance evaluation

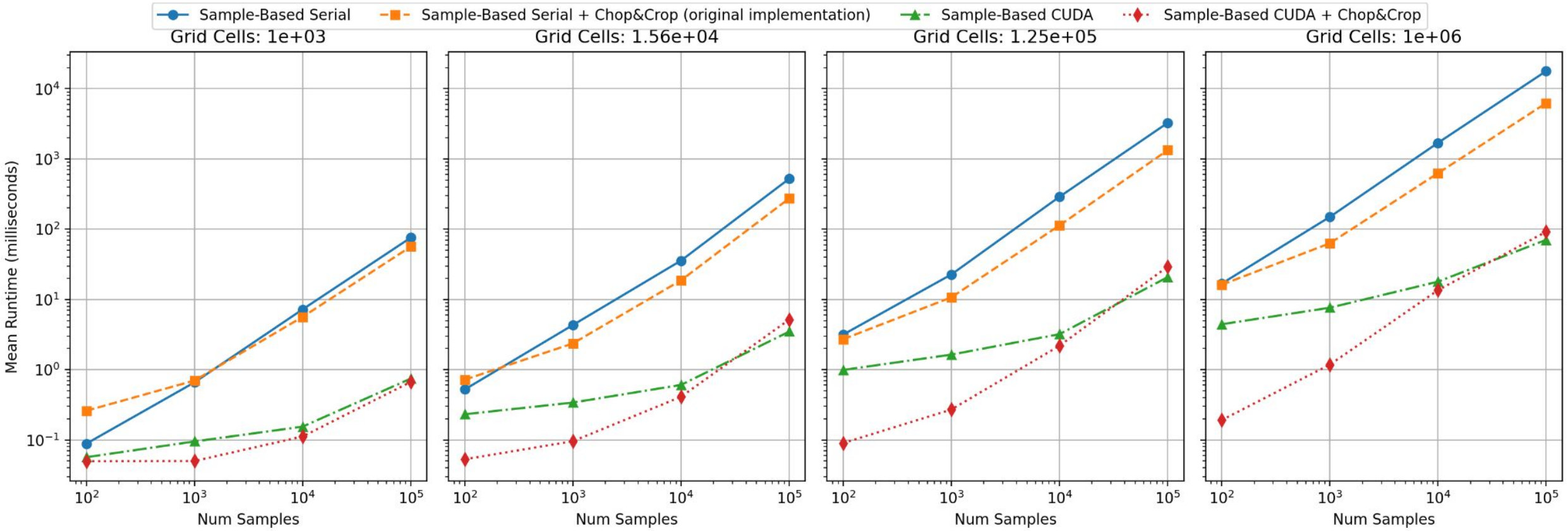
The Journal of Supercomputing Vol. 72, N. 2

2016

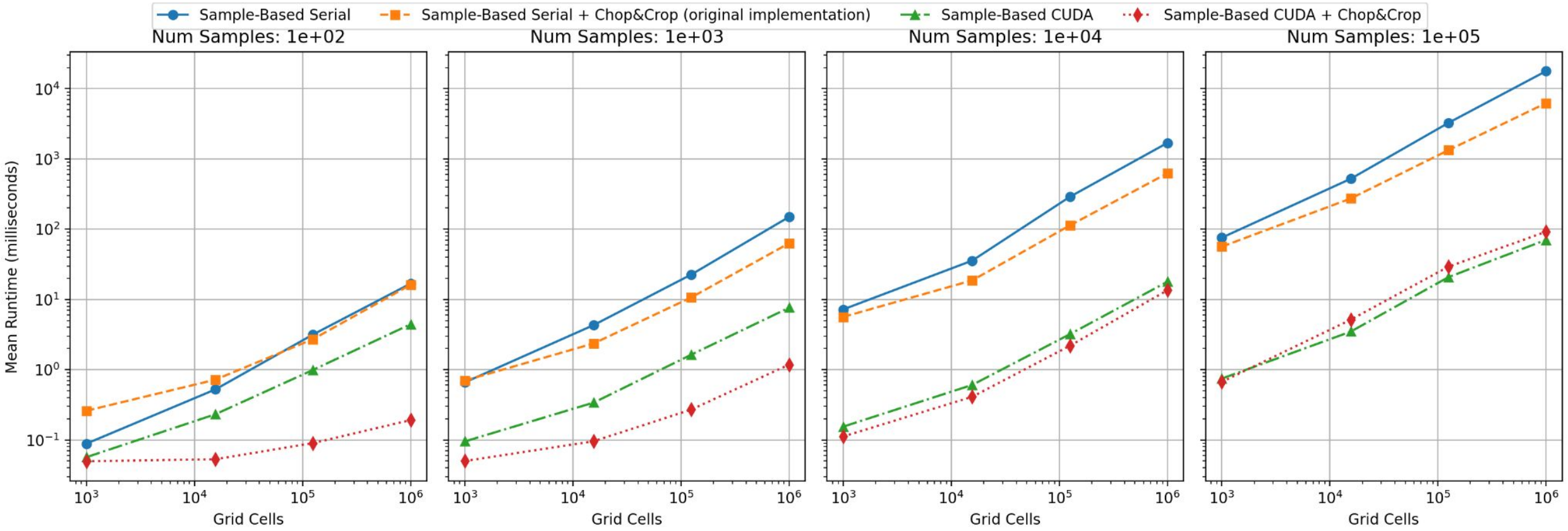
Extra Plots


Mean Speedup by Method (compared to Sample-Based Serial)
Sample-Based Serial
Sample-Based Serial + Chop&Crop (original implementation)
Sample-Based CUDA
Sample-Based CUDA + Chop&Crop
Num Samples
100000
10000
1000
100
1.0x 1.0x 1.0x 1.0x
1.0x 1.0x 1.0x 1.0x
1.0x 1.0x 1.0x 1.0x
1.0x 1.0x 1.0x 1.0x
1.3x 1.9x 2.4x 2.9x
1.3x 1.9x 2.6x 2.7x
0.9x 1.8x 2.1x 2.4x
0.3x 0.7x 1.2x 1.0x
101.4x 150.0x 157.2x 252.4x
46.7x 58.9x 90.4x 94.7x
6.9x 12.7x 13.8x 19.6x
1.6x 2.3x 3.2x 3.8x
113.1x 101.5x 111.6x 193.3x
64.5x 87.0x 132.2x 125.3x
13.2x 45.2x 83.7x 126.7x
1.8x 9.9x 35.1x 87.8x
0.00 0.02 0.12 1.00
Grid Cells (Millions)
Time (log scale)
10^2
10^1
10^0

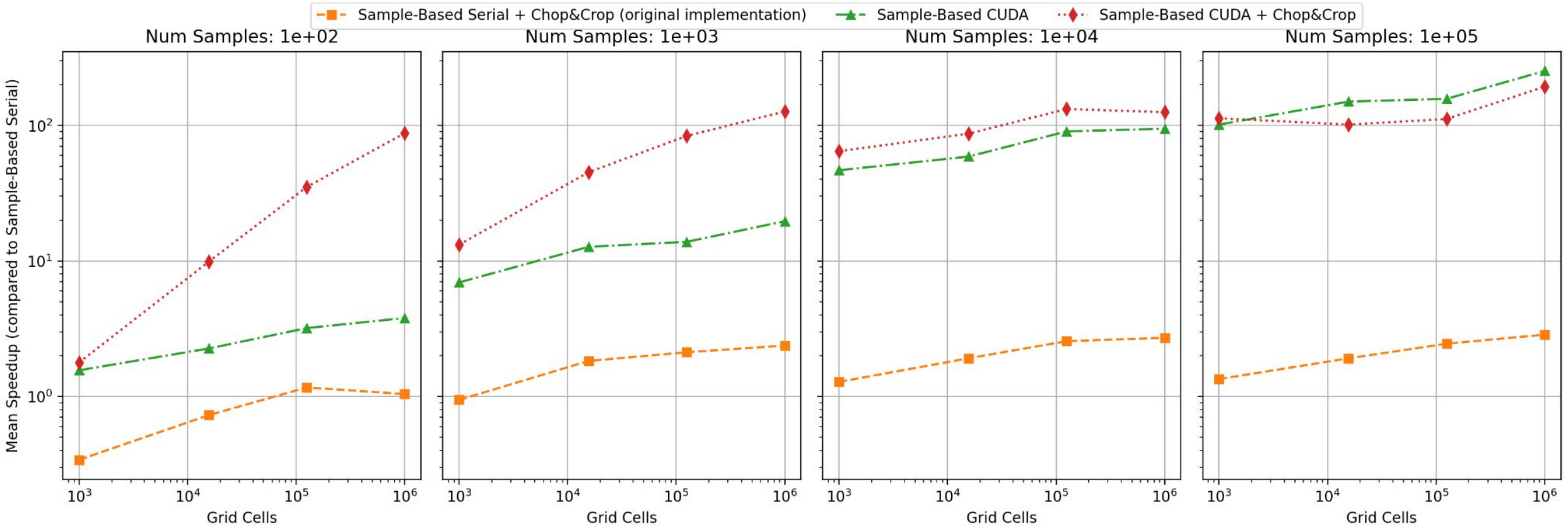
Extra Plots


Sample-Based Serial
Sample-Based Serial + Chop&Crop (original implementation)
Sample-Based CUDA
Sample-Based CUDA + Chop&Crop
Grid Cells: 1e+03
Grid Cells: 1.56e+04
Grid Cells: 1.25e+05
Grid Cells: 1e+06
Mean Runtime (milliseconds)
Num Samples

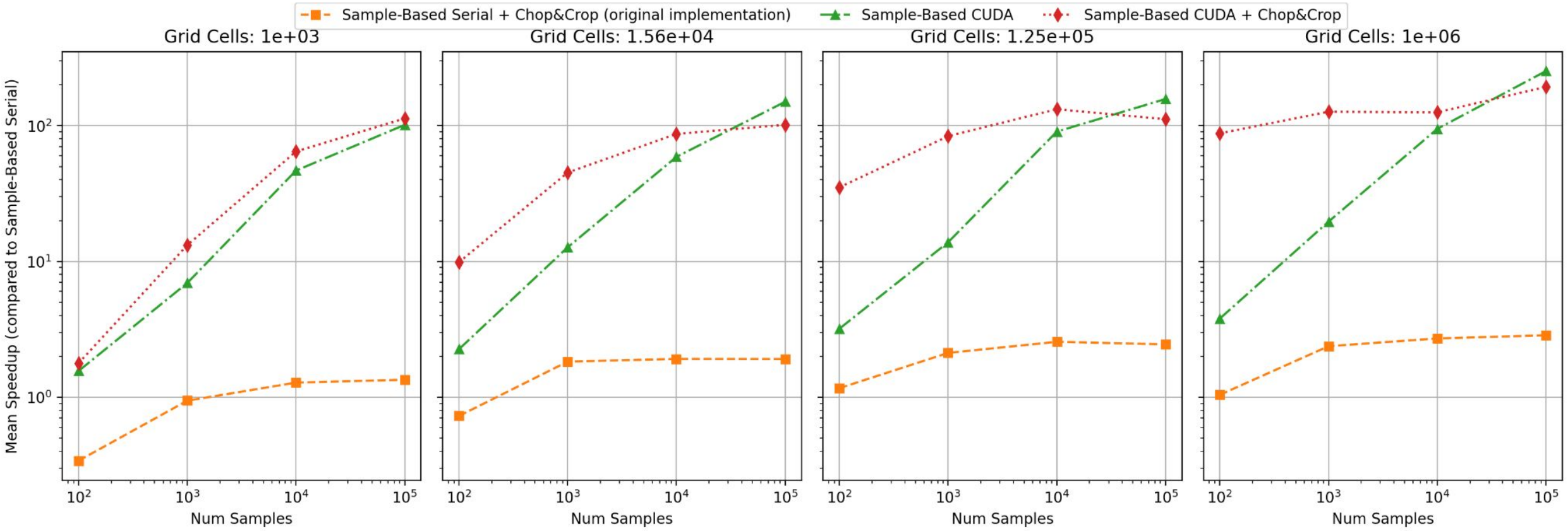
Extra Plots


Sample-Based Serial
Sample-Based Serial + Chop&Crop (original implementation)
Sample-Based CUDA
Sample-Based CUDA + Chop&Crop
Num Samples: 1e+02
Num Samples: 1e+03
Num Samples: 1e+04
Num Samples: 1e+05
Mean Runtime (milliseconds)
Grid Cells

Extra Plots


Sample-Based Serial + Chop&Crop (original implementation)
Sample-Based CUDA
Sample-Based CUDA + Chop&Crop
Num Samples: 1e+02
Num Samples: 1e+03
Num Samples: 1e+04
Num Samples: 1e+05
Mean Speedup (compared to Sample-Based Serial)
10^2
10^1
10^0
10^3
10^4
10^5
10^6
Grid Cells

Extra Plots


Sample-Based Serial + Chop&Crop (original implementation)
Sample-Based CUDA
Sample-Based CUDA + Chop&Crop
Grid Cells: 1e+03
Grid Cells: 1.56e+04
Grid Cells: 1.25e+05
Grid Cells: 1e+06
Mean Speedup (compared to Sample-Based Serial)
Num Samples