P&S Heterogeneous Systems

Algorithmic Improvement and GPU Acceleration of the GenASM Algorithm

Joël Lindegger
ETH Zürich
Spring 2022
24 June 2022
Genome Sequencing & Analysis
Genome Sequencing

- **Genome sequencing:** Enables us to determine the order of the DNA sequence in an organism’s genome
  - Plays a pivotal role in:
    - Personalized medicine
    - Outbreak tracing
    - Understanding of evolution

- **Challenges:**
  - There is no sequencing machine that takes long DNA as an input, and gives the complete sequence as output
  - Sequencing machines extract *small randomized fragments* of the original DNA sequence
Genome Sequencing (cont’d)

Sample Collection

Large DNA molecule

Chopped DNA fragments

Sequencing

Sequenced reads

Preparation

Genome Sequence Analysis

ACGACGTAGCT
ACGTACCCCGT
CTAGGGACCTT
GATACACTGTG
AAAAAAAAAAA
ACGAGCGG GT
Genome Sequencing (cont’d)

Sample Collection

Large DNA molecule

Preparation

Sequencing

Chopped DNA fragments

Sequenced reads

Genome Sequence Analysis

ACGACGTTAAGCT
ACGTACCCCGT
CTAGGACCTT
GATACACTGTG
ACGAGCGGGT
TTTTTTTTAATT
CTAGGGACCTT
Sequencing Technologies

Oxford Nanopore (ONT)

PacBio

Illumina
Current State of Sequencing

*From NIH (https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data)
Current State of Sequencing

Cost per Human Genome

Computation is a bottleneck!

*From NIH (https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data)
Genome Sequencing (cont’d)

Sample Collection

Large DNA molecule

Preparation

Chopped DNA fragments

Sequencing

Sequenced reads

Genome Sequence Analysis

ACGACG TAGCT
ACGTACCCCGT
CTAGGGACCTT
GATACACTGTG
AAAAAAAAAAA
ACGAGCGGGT
TTTTTTTAATT
CTAGGGACCTT
ACGACG TAGCT
Read Mapping, method of aligning the reads against the reference genome in order to detect matches and variations.

De novo Assembly, method of merging the reads in order to construct the original sequence.
**GSA with Read Mapping**

- **Read mapping**: *First key step* in genome sequence analysis (GSA)
  - Aligns *reads* to one or more possible locations within the *reference genome*, and
  - Finds the *matches* and *differences* between the read and the reference genome segment at that location

- Multiple steps of read mapping require *approximate string matching*
  - Approximate string matching (ASM) enables read mapping to account for *sequencing errors* and *genetic variations* in the reads

- Bottlenecked by the *computational power and memory bandwidth* limitations of existing systems
More on Genomics

Accelerating Genome Analysis with FPGAs, GPUs, and New Execution Paradigms

Course Description

A genome encodes a set of instructions for performing some functions within our cells. Analyzing our genomes helps, for example, to determine differences in those instructions (known as genetic variations) from human to human that may cause diseases or different traits. One benefit of knowing the genetic variations is better understanding and diagnosis of diseases and the development of efficient drugs.

Computers are widely used to perform genome analysis using dedicated algorithms and data structures. However, timely analysis of genomic data remains a daunting challenge, due to the complex algorithms and large datasets used for the analysis. Increasing the number of processing cores used for genome analysis decreases the overall analysis time, but significantly escalates the cost of building, maintaining, and cooling such a computing cluster, as well as the power/energy consumed by the cluster. This is a critical shortcoming with respect to both energy production and environmental friendliness. Cloud computing platforms can be used as an alternative to distribute the workload, but transferring the data between the clinic and the cloud poses new privacy and legal concerns.

In this course, we will cover the basics of genome analysis to understand the computational steps of the entire pipeline and find the computational bottlenecks. Students will learn about the existing efforts for accelerating one or more of these steps and will have the chance to carry out a hands-on project to improve these efforts.

https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=bioinformatics
Approximate String Matching
Approximate String Matching

- Sequenced genome may not exactly map to the reference genome due to genetic variations and sequencing errors

```
Reference: AAAATGTTTAGTGCTACCTG
Read: AAAATGTTTACTGCTACTTGT
```

- **Approximate string matching (ASM):**
  - Detect the differences and similarities between two sequences
  - In genomics, ASM is required to:
    - Find the *minimum edit distance* (i.e., total number of differences)
    - Find the *optimal alignment* with a *traceback* step
      - Sequence of matches, substitutions, insertions and deletions, along with their positions
  - Usually implemented as a *dynamic programming (DP)* based algorithm
Arithmetic Dynamic Programming for Approximate String Matching

Commonly-used algorithm for ASM in genomics...

...with quadratic time and space complexity
## Bitvector-Based Dynamic Programming for Approximate String Matching: Bitap Algorithm

| Text           | A     | C     | G     | T     | -     |
|----------------|-------|-------|-------|-------|-------|
| Exact Match    | 1111  | 1111  | 1111  | 1111  | 1111  |
| 1 Edit         | 0110  | 1010  | 1100  | 1110  | 1110  |
| 2 Edits        | 0000  | 0000  | 1000  | 1100  | 1100  |
| 3 Edits        | 0000  | 0000  | 0000  | 1000  | 1000  |
| 4 Edits        | 0000  | 0000  | 0000  | 0000  | 0000  |

- **Two-dimensional**, but different axes
- Entries are **bitvectors** instead of numbers
Bitap Algorithm

**Outer Loop**

For each character of the text (char):

Copy previous R bitvectors as oldR

\[ R[0] = (\text{oldR}[0] \ll 1) \mid \text{PM} [\text{char}] \]

**Inner Loop**

For \( d = 1 \ldots k \):

- deletion \( = \text{oldR}[d-1] \)
- substitution \( = \text{oldR}[d-1] \ll 1 \)
- insertion \( = \text{R}[d-1] \ll 1 \)
- match \( = (\text{oldR}[d] \ll 1) \mid \text{PM} [\text{char}] \)

\[ R[d] = \text{deletion} \& \text{mismatch} \& \text{insertion} \& \text{match} \]

Check MSB of R[d]:

- If 1, no match.
- If 0, match with \( d \) many errors.

**Bitwise Operations**
1. So far we only have the **optimal score**

2. However, we also require the **optimal sequence of edits** ("CIGAR string")

The CIGAR string is obtained through "**traceback**", i.e., following the optimal solution through the table.
GenASM

- Damla Senol Cali, Gurpreet S. Kalsi, Zulal Bingol, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, Anant Nori, Allison Scibisz, Sreenivas Subramoney, Can Alkan, Saugata Ghose, and Onur Mutlu, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis" Proceedings of the 53rd International Symposium on Microarchitecture (MICRO), Virtual, October 2020.
  [Lighting Talk Video (1.5 minutes)]
  [Lightning Talk Slides (pptx) (pdf)]
  [Talk Video (18 minutes)]
  [Slides (pptx) (pdf)]
GenASM Extends Bitap

- Adds traceback

- Performance improvements
  - Windowing heuristic
  - Intra-task parallelism
GenASM’s Traceback

For each character of the text (char):
  Copy previous R bitvectors as oldR
  \( R[0] = (\text{oldR}[0] \ll 1) | \text{PM [char]} \)
  For \( d = 1 \ldots k \):

1. Store these
   - deletion = \( \text{oldR}[d-1] \)
   - substitution = \( \text{oldR}[d-1] \ll 1 \)
   - insertion = \( R[d-1] \ll 1 \)
   - match = \((\text{oldR}[d] \ll 1) | \text{PM [char]}\)

   \( R[d] = \text{deletion} \& \text{mismatch} \& \text{insertion} \& \text{match} \)

Check MSB of \( R[d] \):
  If 1, no match.
  If 0, match with \( d \) many errors.

2. Follow a trail of 0’s in the stored bitvectors
Accelerating GenASM on GPUs
Implementation Attempt #1

- Compile existing C code with CUDA, with minor changes
- One **GPU thread** per sequence **pair**
- Store the DP table in **global memory**

![Graph showing operational intensity versus compute throughput with a note indicating severe memory bound.](image)
Implementation Attempt #2

- Compile existing C code with CUDA, with minor changes
- One **GPU thread** per sequence **pair**
- Store the DP table in **shared memory** has limited capacity
Recall: GenASM’s Traceback

For each character of the text (char):
Copy previous R bitvectors as oldR
R[0] = (oldR[0] << 1) | PM [char]
For d = 1...k:

1. Store these
   deletion = oldR[d-1]
   substitution = oldR[d-1] << 1
   insertion = R[d-1] << 1
   match = (oldR[d] << 1) | PM [char]
   R[d] = deletion & mismatch & insertion & match

Check MSB of R[d]:
   If 1, no match.
   If 0, match with d many errors.

2. Follow a trail of 0’s in the stored bitvectors

---

Senol Cali+, “GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis”, MICRO 2020
GenASM’s Memory Footprint

\[
\text{Footprint} = |\text{text}|^*|\text{pattern}|^*3^*k \text{ bits}
\]

\[\text{For each character of the text (char):}\]
\[\text{Copy previous R bitvectors as oldR}\]
\[R[0] = (\text{oldR}[0] \ll 1) \mid \text{PM [char]}\]
\[\text{For } d = 1..k:\]
\[\text{1. Store these}\]
\[\text{deletion} = \text{oldR}[d-1]\]
\[\text{substitution} = \text{oldR}[d-1] \ll 1\]
\[\text{insertion} = R[d-1] \ll 1\]
\[\text{match} = (\text{oldR}[d] \ll 1) \mid \text{PM [char]}\]
\[R[d] = \text{deletion} \& \text{mismatch} \& \text{insertion} \& \text{match}\]
\[\text{Check MSB of } R[d]:\]
\[\text{If 1, no match.}\]
\[\text{If 0, match with } d \text{ many errors.}\]
\[\text{2. Follow a trail of 0’s in the stored bitvectors}\]

Senol Cali+, “GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis”, MICRO 2020
Memory Footprint of Implementation Attempt #2

Assume

- $|\text{text}| = |\text{pattern}| = k = 64$
- One warp of 32 threads

Footprint

$= 64 \times 64 \times 3 \times 64 \text{ bits} \times 32$

$= 96 \text{KiB} \times 32$

$= 3 \text{MiB}$

Shared memory capacity of the NVIDIA A6000 GPU:

99KiB per SM
Implementation Attempt #2

- Compile existing C code with CUDA, with minor changes
- One **GPU thread** per sequence **pair**
- Store the DP table in **shared memory** has **limited capacity**

**Fails**
Because the DP table does not fit into shared memory
Implementation Attempt #3

- Rewrite code for GPUs
- **64 threads** in a thread block cooperate per seq. pair
- Store the DP table in **shared memory**
### Mapping Threads to DP Cells

| Text          | A  | C  | G  | T  | -  |
|---------------|----|----|----|----|----|
| **Exact Match** | 1111 | 1111 | 1111 | 1111 | 1111 |
| **1 Edit**    | 0110 | 1010 | 1100 | 1110 | 1110 |
| **2 Edits**   | 0000 | 0000 | 1000 | 1100 | 1100 |
| **3 Edits**   | 0000 | 0000 | 0000 | 1000 | 1000 |
| **4 Edits**   | 0000 | 0000 | 0000 | 0000 | 0000 |

Thread 0  Thread 1  Thread 2  Thread 3
Memory Footprint of Implementation Attempt #3

Assume

- $|\text{text}| = |\text{pattern}| = k = 64$
- 1 thread block

Footprint

$= 64 \times 64 \times 3 \times 64 \text{ bits} \times 1$
$= 96\text{KiB} \times 1$
$= 96\text{KiB}$

Shared memory capacity of the NVIDIA A6000 GPU: 99KiB per SM
Due to the memory footprint and shared memory capacity, at most 1 thread block of 64 threads can be active per SM. This is called “low occupancy.”

Occupancy should be high for good performance. Simultaneous multithreading hides instruction latencies, but requires multiple active warps to work.
Latency Hiding and Occupancy

- **FGMT** can hide long latency operations (e.g., memory accesses)
- **Occupancy**: ratio of active warps to the maximum number of warps per GPU core

---

[Video Link](https://youtu.be/BYkPHLaFATM)
Implementation Attempt #3

- Rewrite code for GPUs
- **64 threads** in a thread block *cooperate* per seq. pair
- Store the DP table in *shared memory*

Low performance

Due to low occupancy
Scrooge

- Joël Lindegger, Damla Senol Cali, Mohammed Alser, Juan Gomez-Luna, and Onur Mutlu, "Algorithmic Improvement and GPU Acceleration of the GenASM Algorithm" Proceedings of the 21st IEEE International Workshop on High Performance Computational Biology (HiCOMB), Virtual, May 2022. [Slides (pptx) (pdf)]
## Motivation

- **Pairwise sequence alignment** is a bottleneck in genomic pipelines (e.g., read mapping)
- **GenASM** is a state-of-the-art sequence aligner, its hardware implementation is up to 10,000x faster than prior software aligners and draws 500x less power

## Problem

Three inefficiencies in the GenASM algorithm limit its throughput and energy efficiency:
1. It has a large memory footprint
2. It has a high bandwidth pressure
3. It does some unnecessary work

## Goal

Enable **fast** and **efficient** implementations of GenASM for CPUs, GPUs, and ASICs

## Scrooge

Three novel algorithmic improvements to the GenASM algorithm:
- **SENE** and **DENT** reduce the memory footprint and data movement of GenASM
- **Early Termination** eliminates unnecessary work

High-performance **CPU** and **GPU** implementations

## Results

- **12-24x memory footprint reduction**
- **1.9x speedup** over GenASM on a recent CPU (Xeon Gold 5118)
- **5.9x speedup** over GenASM on a recent GPU (NVIDIA A6000)
- Similar improvements to be expected for ASICs
Algorithmic Improvement 1: Store Entries, Not Edges (SENE)

**GenASM**
Stores **3 edges** per table entry for traceback

| Text    | A | C | G | T | - |
|---------|---|---|---|---|---|
| Exact Match |   |   |   |   |   |
| 1 Edit   |   |   |   |   |   |
| 2 Edits  |   |   |   |   |   |
| 3 Edits  |   |   |   |   |   |
| 4 Edits  |   |   |   |   |   |

**Scrooge**
Stores table entries directly, edges are **regenerated**

| Text         | A | C | G | T | - |
|--------------|---|---|---|---|---|
| Exact Match  | 1111 | 1111 | 1111 | 1111 | 1111 |
| 1 Edit       | 0110 | 1010 | 1100 | 1110 | 1110 |
| 2 Edits      | 0000 | 0000 | 1000 | 1100 | 1100 |
| 3 Edits      | 0000 | 0000 | 0000 | 1000 | 1000 |
| 4 Edits      | 0000 | 0000 | 0000 | 0000 | 0000 |

**SENE** results in a **3x reduction** in memory footprint and data movement

Lindegger+, "Algorithmic Improvement and GPU Acceleration of the GenASM Algorithm", HiCOMB 2022
Algorithmic Improvement 2: 
Discard Entries Not used by Traceback (DENT)

GenASM’s **traceback** does **not cross** the entire table
Scrooge **discards** entries that are **never reached**

| Text       | A   | C   | G   | T   | -   |
|------------|-----|-----|-----|-----|-----|
| Exact Match| 11X | 11X | 1111| 1111| 11  |
| 1 Edit     | 010 | 010 | 1110| 1110| 10  |
| 2 Edits    | 00X | 00X | 1000| 1100| 11  |
| 3 Edits    | 00X | 00X | 0100| 0100| 11  |
| 4 Edits    | 00X | 00X | 0010| 0010| 11  |

**DENT** results in a **4x reduction** in memory footprint and data movement

Lindegger+, "Algorithmic Improvement and GPU Acceleration of the GenASM Algorithm", HiCOMB 2022
Implementation Attempt #4 – Scrooge

- **64 threads** in a thread block cooperate per seq. pair
- Store the DP table in **shared memory**
- Algorithmically **reduced memory footprint**

---

**GenASM**
- Stores **3 edges** per table entry for traceback
- SENE Improvement

**Scrooge**
- Stores table entries directly, edges are **regenerated**
- SENE results in a **3x reduction** in memory footprint and data movement

GenASM’s traceback does not cross the entire table. **Scrooge discards** entries that are never reached.

**SENEDENT** results in a **4x reduction** in memory footprint and data movement

---

**SENEDENT** results in a **12x reduction** in memory footprint and data movement

---

Lindegger+, "Algorithmic Improvement and GPU Acceleration of the GenASM Algorithm", HiCOMB 2022
Scrooge’s Memory Footprint

Assume
- $|\text{text}| = |\text{pattern}| = k = 64$
- 12 thread blocks

Footprint
\[
\frac{64}{2} \times \frac{64}{2} \times 1 \times 64 \text{ bits} \times 12 \\
= 8\text{KiB} \times 12 \\
= 96\text{KiB}
\]

Shared memory capacity of the NVIDIA A6000 GPU: 99KiB per SM

Lindegger+, "Algorithmic Improvement and GPU Acceleration of the GenASM Algorithm", HiCOMB 2022
Implementation Attempt #4 – Scrooge

- 64 threads in a thread block cooperate per seq. pair
- Store the DP table in shared memory
- Algorithmically reduced memory footprint

High performance
Thanks to high occupancy

SENE+DENT results in a 12x reduction in memory footprint and data movement

Lindegger+, "Algorithmic Improvement and GPU Acceleration of the GenASM Algorithm", HiCOMB 2022
Scaling with Threadblocks

Lindegger+, "Algorithmic Improvement and GPU Acceleration of the GenASM Algorithm“, HiCOMB 2022
Generating Pattern Bitmasks
Pattern Bitmasks

- GenASM requires pre-computing "pattern bitmasks"
  - One-hot (or one-cold) encoding of the pattern string

Pattern Bitmasks:

```
CTGA
PM(A) = 1110
PM(C) = 0111
PM(G) = 1101
PM(T) = 1011
```
Pattern Bitmasks – Serial Implementation

Pattern Bitmasks:

- CTGA
- PM(A) = 1110
- PM(C) = 0111
- PM(G) = 1101
- PM(T) = 1011

```c
__device__ PM_t generatePatternBitmask(int m, char *pattern) {
    PM_t pm;
    pm[A] = ONES;
    pm[C] = ONES;
    pm[G] = ONES;
    pm[T] = ONES;
    for (int j = 0; j < m; j++) {
        pm[pattern[j]] &= SINGLE_ZERO_AT(j);
    }
    return pm;
}
```

Initialize to all 1s

For each char in pattern

Set a 0

No parallelism
Only one thread per block does useful work
**Pattern Bitmasks – Parallel Implementation**

Pattern Bitmasks:

- CTGA
- PM(A) = 1110
- PM(C) = 0111
- PM(G) = 1101
- PM(T) = 1011

```c
__device__ PM_t generatePatternBitmask(int m, char *pattern) {

    PM_t pm;
    pm[A] = ONES;
    pm[C] = ONES;
    pm[G] = ONES;
    pm[T] = ONES;

    int j = threadIdx.x;
    pm[pattern[j]] &= SINGLE_ZERO_AT(j);

    for (int offset = 16; offset > 0; offset /= 2) {
        // bitwise AND parallel reduction
        ...
    }

    return pm;
}
```

**Initialize to all 1s**

**Each thread sets a 0**

**Merge results**

**Large amount parallelism**

Every thread in the block does useful work
Warp Shuffle Functions

- Built-in **warp shuffle functions** enable threads to share data with other threads in the same warp
  - Faster than using shared memory and __syncthreads() to share across threads in the same block
- Variants:
  - __shfl_sync(mask, var, srcLane)
    - Direct copy from indexed lane
  - __shfl_up_sync(mask, var, delta)
    - Copy from a lane with lower ID relative to caller
  - __shfl_down_sync(mask, var, delta)
    - Copy from a lane with higher ID relative to caller
  - __shfl_xor_sync(mask, var, laneMask)
    - Copy from a lane based on bitwise XOR of own lane ID

https://youtu.be/Xp0HHpcDwUc
Resolving Data Dependencies
Recall: Mapping Threads to DP Cells

| Text       | A  | C  | G  | T  | -  |
|------------|----|----|----|----|----|
| Exact Match| 1111| 1111| 1111| 1111| 1111 |
| 1 Edit     | 0110| 1010| 1100| 1110| 1110 |
| 2 Edits    | 0000| 0000| 1000| 1100| 1100 |
| 3 Edits    | 0000| 0000| 0000| 1000| 1000 |
| 4 Edits    | 0000| 0000| 0000| 0000| 0000 |

Thread 0  Thread 1  Thread 2  Thread 3
## Data Dependencies

| Text         | A | C | G | T | - |
|--------------|---|---|---|---|---|
| **Exact Match** | 1111 | 1111 | 1111 | 1111 | 1111 |
| **1 Edit**   | 0110 | 1010 | 1100 | 1110 | 1110 |
| **2 Edits**  | 0000 | 0000 | 1000 | 1100 | 1100 |
| **3 Edits**  | 0000 | 0000 | 0000 | 1000 | 1000 |
| **4 Edits**  | 0000 | 0000 | 0000 | 0000 | 0000 |

Thread 0: 
- Intra-Thread Data Dependency

Thread 1: 
- Inter-Thread Data Dependency
# Data Dependencies

| Text        | A       | C       | G       | T       | -       |
|-------------|---------|---------|---------|---------|---------|
| **Exact Match** | 1111    | 1111    | 1111    | 1111    | 1111    |
| **1 Edit**   | 0110    | 1010    | 1100    | 1110    | 1110    |
| **2 Edits**  | 0000    | 0000    | 1000    | 1100    | 1100    |
| **3 Edits**  | 0000    | 0000    | 0000    | 1000    | 1000    |
| **4 Edits**  | 0000    | 0000    | 0000    | 0000    | 0000    |

Thread 0 | Thread 1 | Thread 2 | Thread 3 | Thread 4

- Intra-Thread Data Dependency
- Inter-Thread Data Dependency

Iteration 0
Iteration 1
Iteration 2
Iteration 3
Resolving Data Dependencies

- **Memory accesses** can resolve all dependencies
  - But have undesirable **latency** and **bandwidth**
- **Register accesses** are preferred
  - Works **trivially** for vertical **intra-thread** dependencies
  - Does it also work for horizontal **inter-thread** dependencies?
Resolving Data Dependencies (cont’d)

- How to resolve **inter-thread** dependencies with registers?
  - Warp **shuffle** instructions!

**B.22.1. Synopsis**

```c
T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);
T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);
```

**B.22.2. Description**

The `__shfl_sync()` intrinsics permit exchanging of a variable between threads within a warp without use of shared memory. The exchange occurs simultaneously for all **active** threads within the warp (and named in `mask`), moving 4 or 8 bytes of data per thread depending on the type.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions
More About Warp Shuffle Functions

Warp Shuffle Functions

- **Built-in warp shuffle functions** enable threads to share data with other threads in the same warp
  - Faster than using shared memory and \_syncthreads() to share across threads in the same block

- **Variants:**
  - \_shfl_sync(mask, var, srcLane)
    - Direct copy from indexed lane
  - \_shfl_up_sync(mask, var, delta)
    - Copy from a lane with lower ID relative to caller
  - \_shfl_down_sync(mask, var, delta)
    - Copy from a lane with higher ID relative to caller
  - \_shfl_xor_sync(mask, var, laneMask)
    - Copy from a lane based on bitwise XOR of own lane ID

[YouTube Video: More About Warp Shuffle Functions](https://youtu.be/Xp0HHpcDwUc)
Conclusion

Before: 1,800 alignments/second

After: 25,000 alignments/second

- Shared Memory
- Thread Cooperative Implementation
- Algorithmic Improvements
- Bitmasks with parallel reduction
- Data dependencies with warp shuffle

Optimizing GPU code requires a holistic view

Ignoring any of the above aspects would ruin the performance of the program
More P&S Courses: SSDs, Memory, Bioinformatics…

- Understanding and Improving Modern DRAM Performance, Reliability, and Security with Hands-On Experiments
- Designing and Evaluating Memory Systems and Modern Software Workloads with Ramulator
- Accelerating Genome Analysis with FPGAs, GPUs, and New Execution Paradigms
- Genome Sequencing on Mobile Devices
- Understanding and Designing Modern NAND Flash-Based Solid-State Drives (SSDs)
- Intelligent Architectures using Hardware/Software Cooperative Techniques

https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=start
More Resources: Onur Mutlu Lectures

- All P&S courses
- Digital Design and CompArch course
- Advanced CompArch course
- Seminar in CompArch

https://www.youtube.com/c/OnurMutluLectures/playlists
Algorithmic Improvement and GPU Acceleration of the GenASM Algorithm

Joël Lindegger
ETH Zürich
Spring 2022
24 June 2022