

GPU-BASED ACCELARATION OF DENOISING PYROSEQUENCED AMPLICONS

Byunghan Lee and Sungroh Yoon

Advanced Computing Laboratory
Korea University

Outline

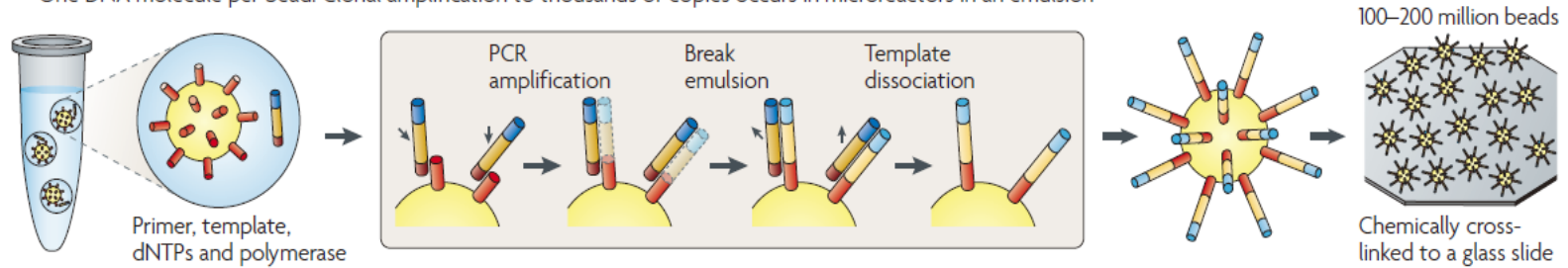
- **Introduction**
 - NGS, Pyrosequencing
 - Pyrosequencing Noise
 - Removing Pyrosequencing Noise
- **Methods**
 - AmpliconNoise Profiling
 - GPGPU, CUDA
 - Parallelize AmpliconNoise
 - Optimization
- **Results & Discussion**
- **Summary**

Introduction

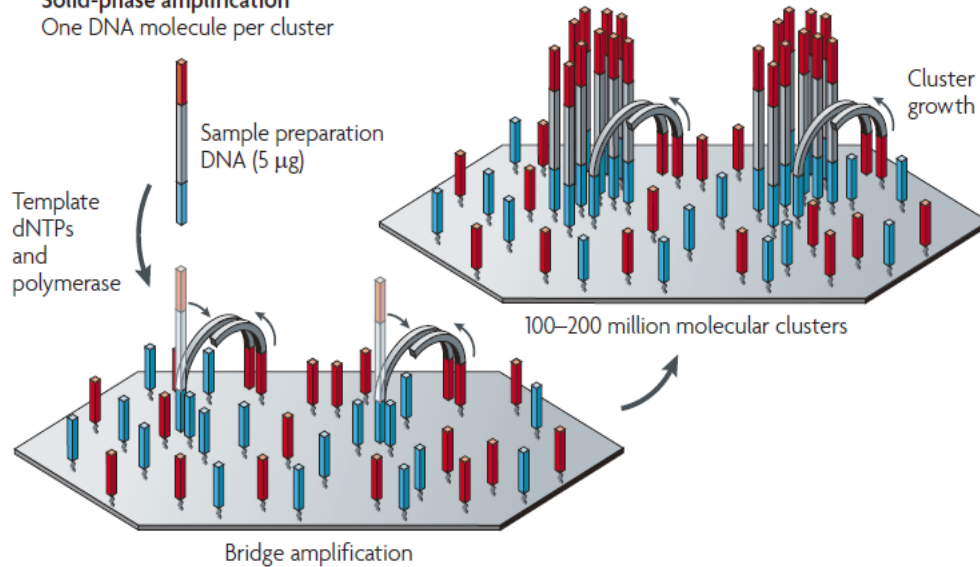
Next Generation Sequencing (NGS)

a Roche/454, Life/APG, Polonator Emulsion PCR

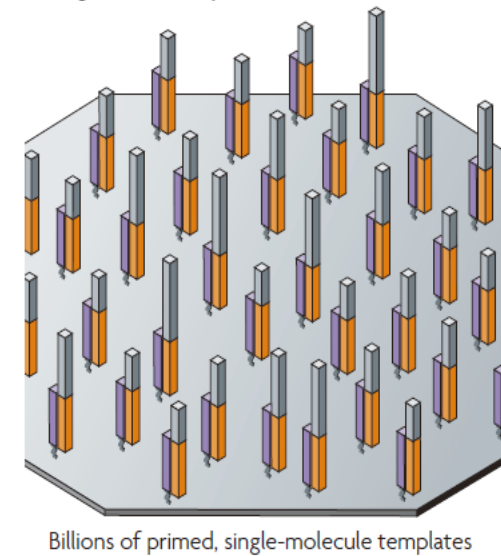
One DNA molecule per bead. Clonal amplification to thousands of copies occurs in microreactors in an emulsion



b Illumina/Solexa Solid-phase amplification One DNA molecule per cluster



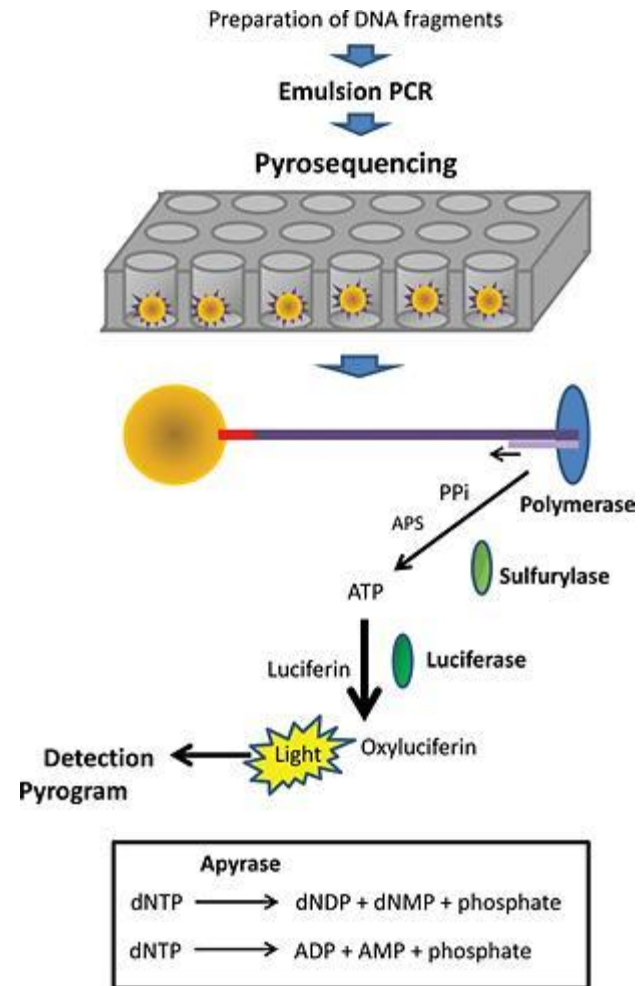
c Helicos BioSciences: one-pass sequencing Single molecule: primer immobilized



Source: Metzker, M. L., **DNA sequencing with chain-terminating inhibitors.**

Pyrosequencing

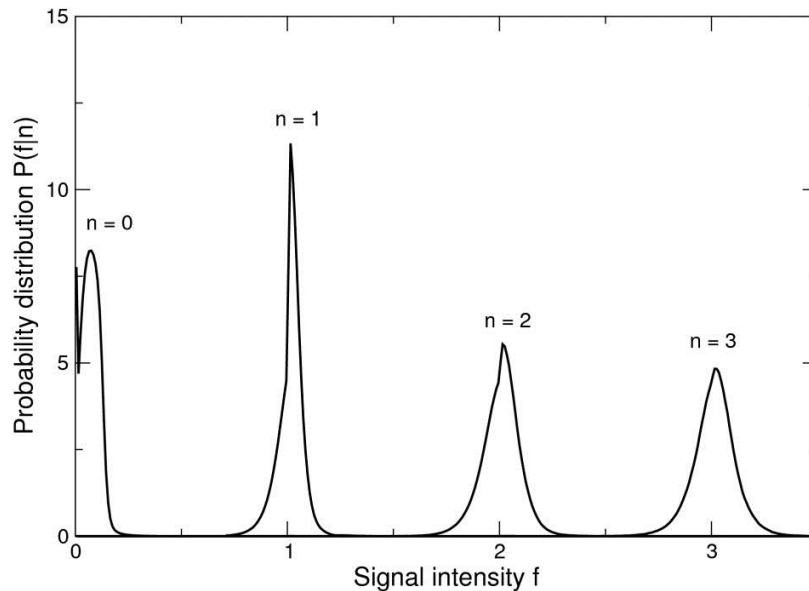
- Roche 454



Source: Siqueira et al., **Pyrosequencing as a tool for better understanding of human microbiomes.**

Pyrosequencing Noise (1/3)

- Flowgram Signal Intensity Distributions



Source: Quince et al., **Removing Noise From Pyrosequenced Amplicons**.

Raw data

0.03	1.03	0.09	0.12	1.49	0.09	0.09	1.01
------	------	------	------	------	------	------	------

T C A G

The result read (wrong)

	C			T			G
--	---	--	--	---	--	--	---

The true read (correct)

	C			TT			G
--	---	--	--	----	--	--	---

- The observed light intensities do not perfectly match the homopolymer lengths.

Pyrosequencing Noise (2/3)

- **PCR per Base Error Probabilities**

- Erroneous nucleotide transition during PCR amplification.
- The nucleotide transition probabilities were calculated by comparing all reads with pyrosequencing noise removed from the three 'Even' V2 data sets with the known control sequences[1].

	A	C	T	G
A	0.9995	7.2e-6	7.7e-6	5.1e-4
C	1.1e-05	0.9996	4.1e-4	2.1e-6
T	9.0e-6	5.7e-4	0.9994	1.4e-5
G	3.5e-4	3.2e-6	2.1e-5	0.9996

The result read (wrong)

C A C A A G G C

The true read (correct)

C A T A A G G C



Source: Quince et al., **Removing Noise From Pyrosequenced Amplicons.**

Pyrosequencing Noise (3/3)

- **Chimeras**

- Sequences that are composed of two or more true sequences.
- Chimeras are generated when incomplete extension occurs during the PCR process.
- These generated sequences are quite different from either parent.

- We are focusing on the first two sources of error (shown on slides 6 & 7).

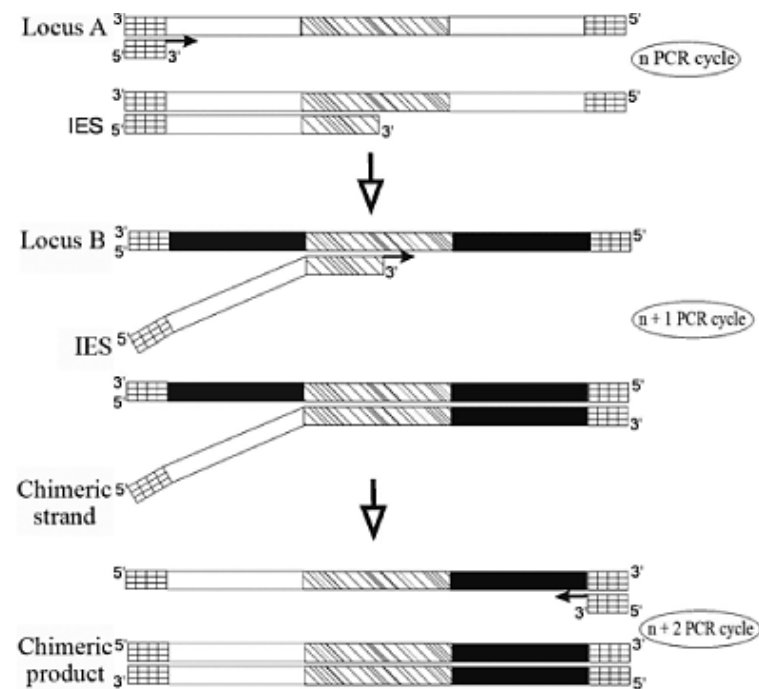
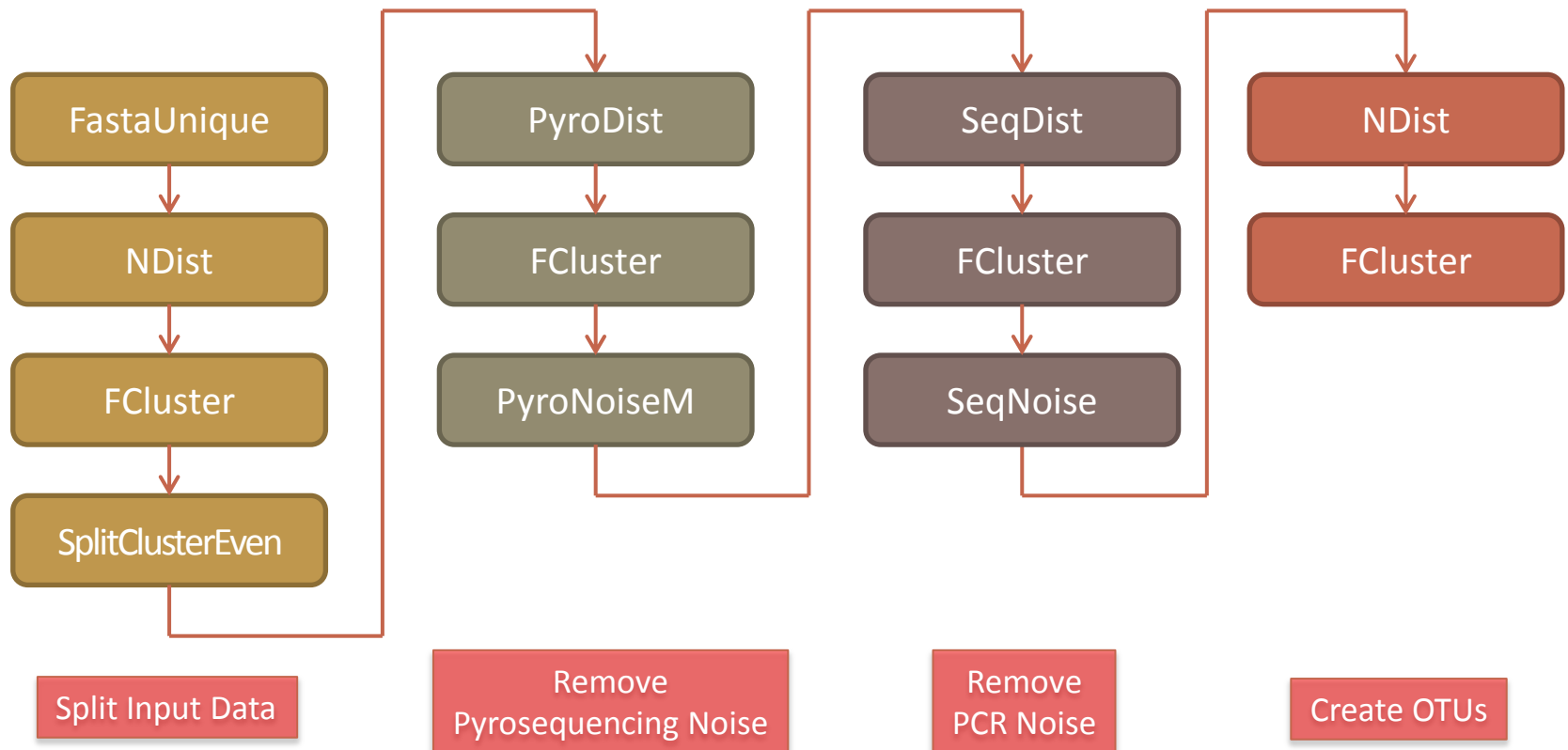


Figure 1 - Schematic representation of three successive PCR cycles related to formation of chimeric products. The illustrations represent two isolated microsatellite loci with the same repeat motif (light hatched region) but different flanking regions. White boxes show flanking regions of locus A, while black boxes show the flanking regions of locus B. Adapters functioning as primers are shown as checkered boxes. In the n cycle, an incompletely extended strand (IES) was produced from locus A, which ends in the repeat and serves as a primer for locus B in the $n + 1$ cycle. A chimeric strand is generated, which becomes double-stranded in the next cycle ($n + 2$) and which can be amplified in the following cycles (not represented).

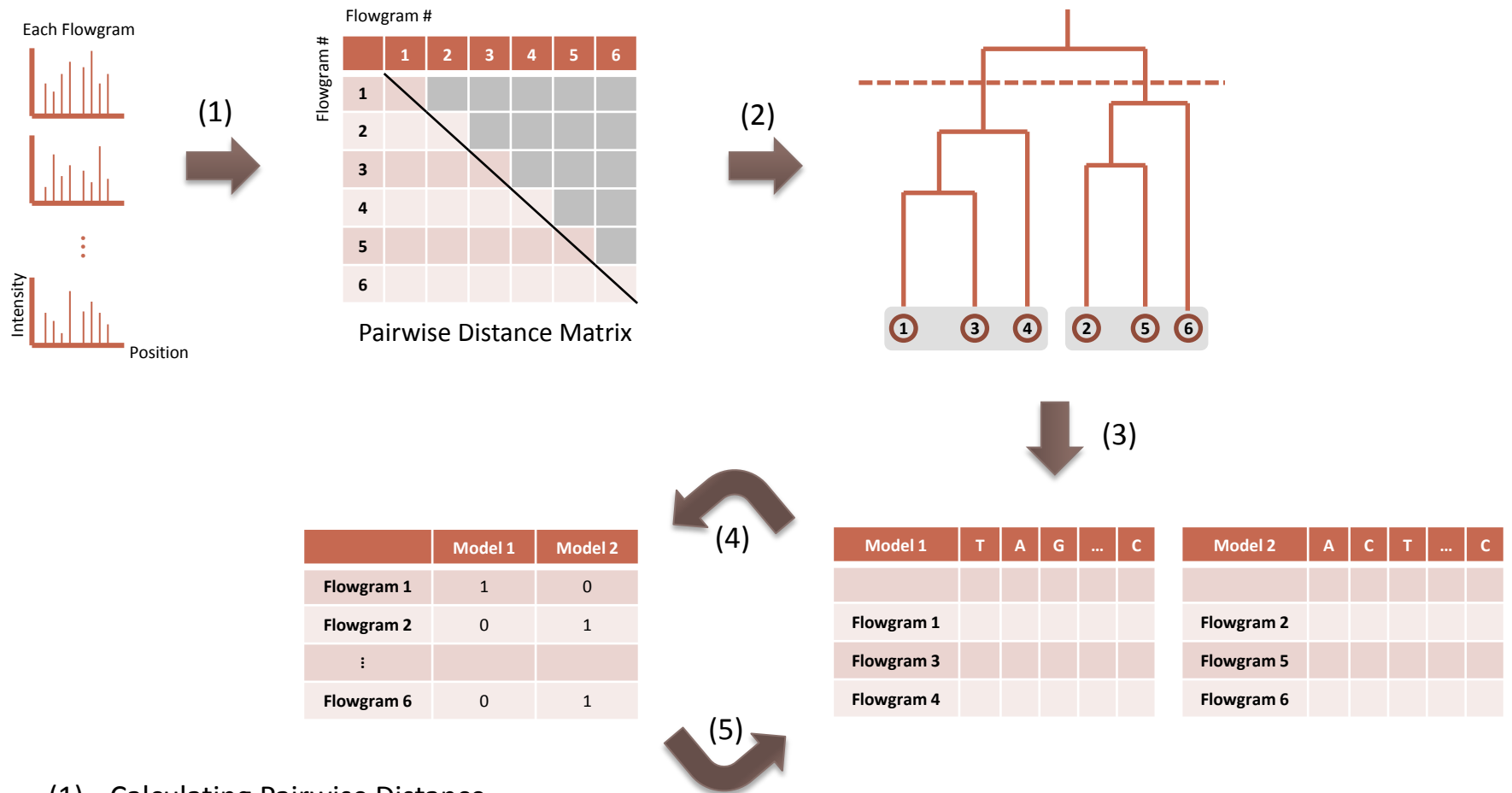
Source: Roratto et al., **PCR-mediated recombination in development of microsatellite markers: mechanism and implications.**

AmpliconNoise

- Collection of programs for the removal of noise from 454 sequenced PCR amplicons developed by Quince *et al.* (2011).
- Consists of **12 steps** in dealing with Titanium reads,

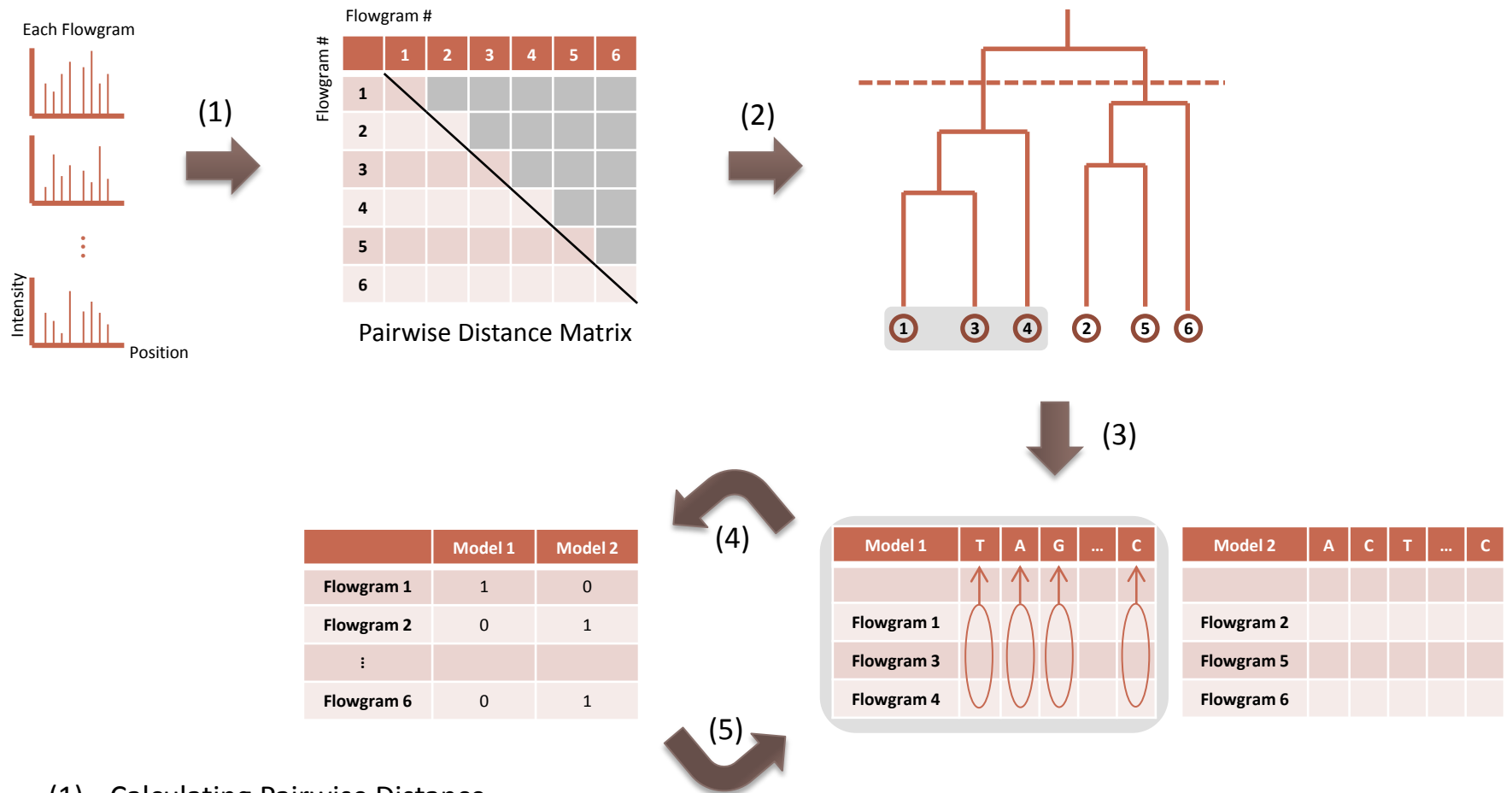


Removing Noise (1/4)



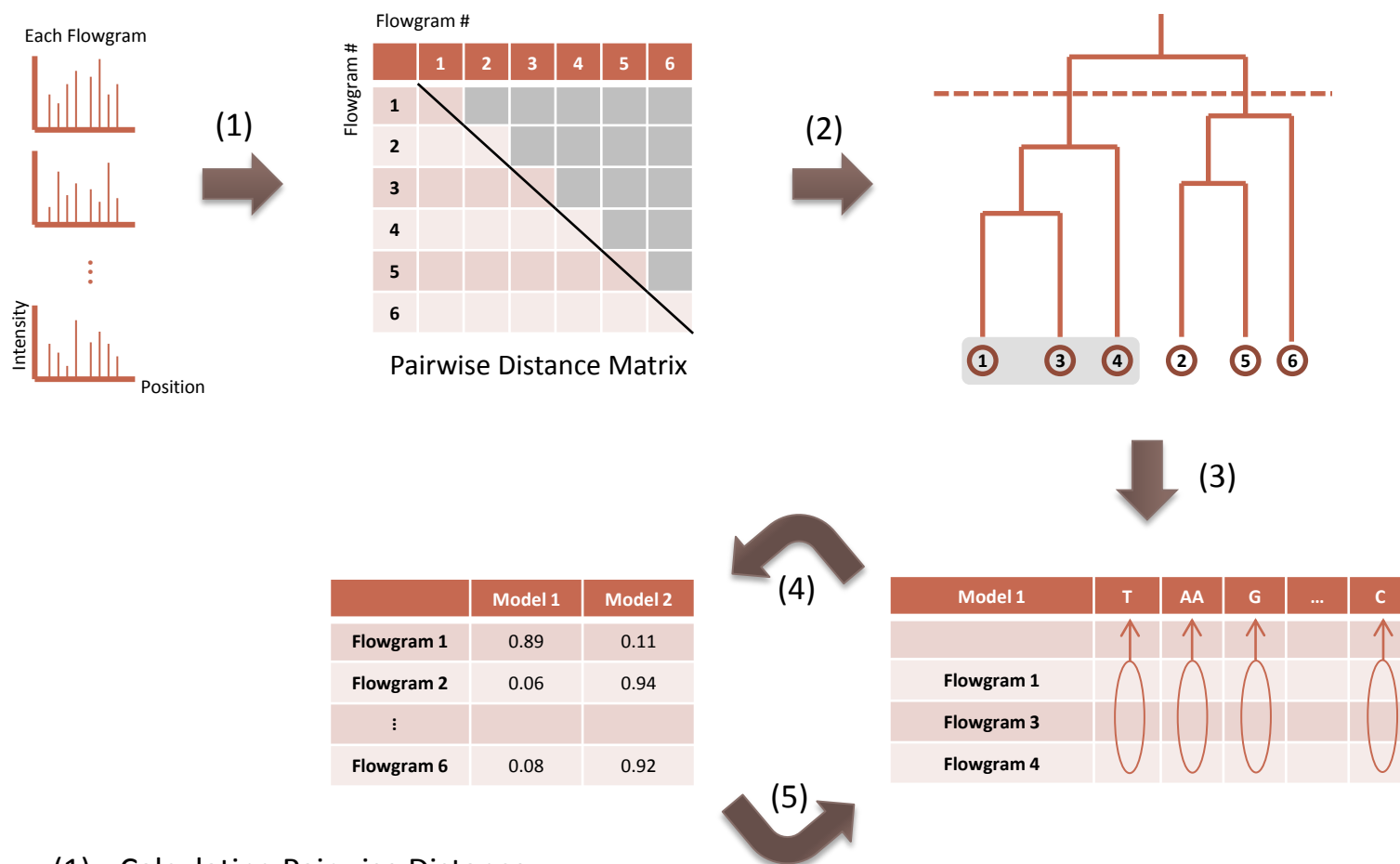
- (1) Calculating Pairwise Distance
- (2) Hierarchical Clustering
- (3) Creating Model
- (4) E Step
- (5) M Step

Removing Noise (1/4)



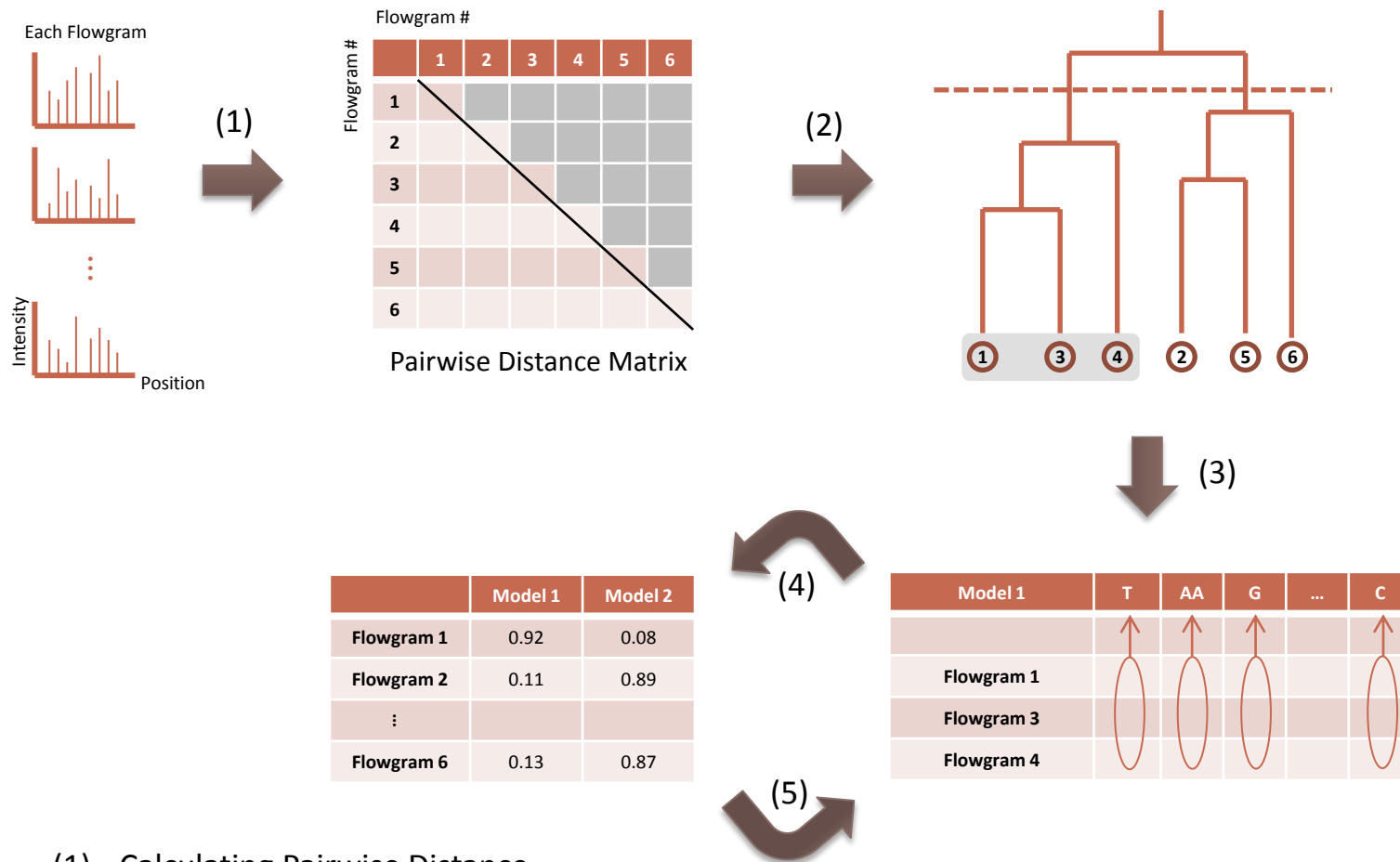
- (1) Calculating Pairwise Distance
- (2) Hierarchical Clustering
- (3) Creating Model
- (4) E Step
- (5) M Step

Removing Noise (1/4)



- (1) Calculating Pairwise Distance
- (2) Hierarchical Clustering
- (3) Creating Model
- (4) E Step
- (5) M Step

Removing Noise (1/4)



- (1) Calculating Pairwise Distance
- (2) Hierarchical Clustering
- (3) Creating Model
- (4) E Step
- (5) M Step

Removing Noise (2/4)

- **Removing Sequencing Noise[1, 2, 4]**

- Define distances for given flowgram $\bar{f} = (f_1, \dots, f_M)$.

$$\begin{aligned}d'(\bar{f}, \bar{U}) &= -\log \left(\prod_{i=1}^M P(f_i | u_i = n) \right) / M \\&= \sum_{i=1}^M -\log [P(f_i | u_i = n)] / M && (\bar{U}: \text{perfect flowgram, } M: \text{flowgram length}) \\&= \sum_{i=1}^M d(f_i | u_i = n) / M\end{aligned}$$

- To define the likelihood, the density of the observed flowgrams are:

$$F(\bar{f}_i | \bar{S}_j \rightarrow \bar{U}_j) = \frac{\exp[-d'(\bar{f}_i, \bar{U}_j) / \sigma_p]}{\sigma_p} \quad (S: \text{modeled sequence, } \sigma_p: \text{characteristic cluster size})$$

- The likelihood of the complete data set D is then:

$$L(D | L; \tau_1, \dots, \tau_L; \bar{S}_1, \dots, \bar{S}_L) = \prod_{i=1}^N \left[\sum_{j=1}^L \tau_j F(\bar{f}_i | \bar{S}_j) \right]$$

(D: observed data set, L: # models, N: # flowgrams, τ : relative frequency)

Removing Noise (3/4)

- **Removing Sequencing Noise[1, 2, 4]**

- Using an expectation-maximization algorithm to infer the true sequences and their frequencies.

- M step: Find the perfect flowgram with the smallest total distance to all the reads.

$$\bar{U}_j = \bar{Q}_l \text{ where } l = \min_k \left[\sum_{i=1}^N \hat{z}_{i,j} d'(\bar{f}_i, \bar{Q}_k) \right] \quad (Q: \text{unique perfect flowgram})$$

- Calculate new distances $d'(\bar{f}_i, \bar{U}_j)$.
- E step: Calculate new $\hat{z}_{i,j}$ as the conditional probabilities that sequence j generated flowgram i .

$$\hat{z}_{i,j} = \frac{\tau_i \exp\left(-\frac{d'(\bar{f}_i, \bar{U}_j)}{\sigma_p}\right)}{\sum_{k=1}^L \tau_k \exp\left(-\frac{d'(\bar{f}_i, \bar{U}_k)}{\sigma_p}\right)}$$

- Repeat until convergence.

Removing Noise (4/4)

- **Removing PCR Point Error[1, 2, 4]**

$$\begin{aligned}e(\bar{r}, \bar{S}) &= -\log [P(\bar{r}|\bar{S})] \\&= -\log \left[\prod_{l=1}^M P(r^l = m | s^l = n) \right] / A \\&= \sum_{l=1}^A -\log [P(r^l = m | s^l = n)] / A\end{aligned}$$

- (e : distance, \bar{r} : given read, \bar{S} : true sequence, A : alignment length)

	A	C	T	G
A	0.9995	7.2e-6	7.7e-6	5.1e-4
C	1.1e-05	0.9996	4.1e-4	2.1e-6
T	9.0e-6	5.7e-4	0.9994	1.4e-5
G	3.5e-4	3.2e-6	2.1e-5	0.9996

- Use **mixture model** to cluster the sequences.
- Reads are assumed to be distributed as exponentially decaying functions of their sequence error corrected distance from true sequences.
- Maximum likelihood fit of the mixture model can be obtained using an **Expectation-Maximization** (EM) algorithm.
- Identical to previous noise remover step except that in previous step, distance is defined between flowgrams, but here it is defined between sequences.

Methods

Profiling (1/3)

- The tests were run on the following hardware:
 - CPU: 2x Intel Xeon E5405 @ 2.00Ghz
 - RAM: 16GB
 - OS: Ubuntu Server 10.04.3 LTS 64bit
- The tests were run on the following data set[1]:
 - Generated by pyrosequencing a mixture of 91 full length 16S rRNA clones obtained from an Arctic soil sample.
 - # of Reads: 62,873

Profiling (2/3)

- Using MPI

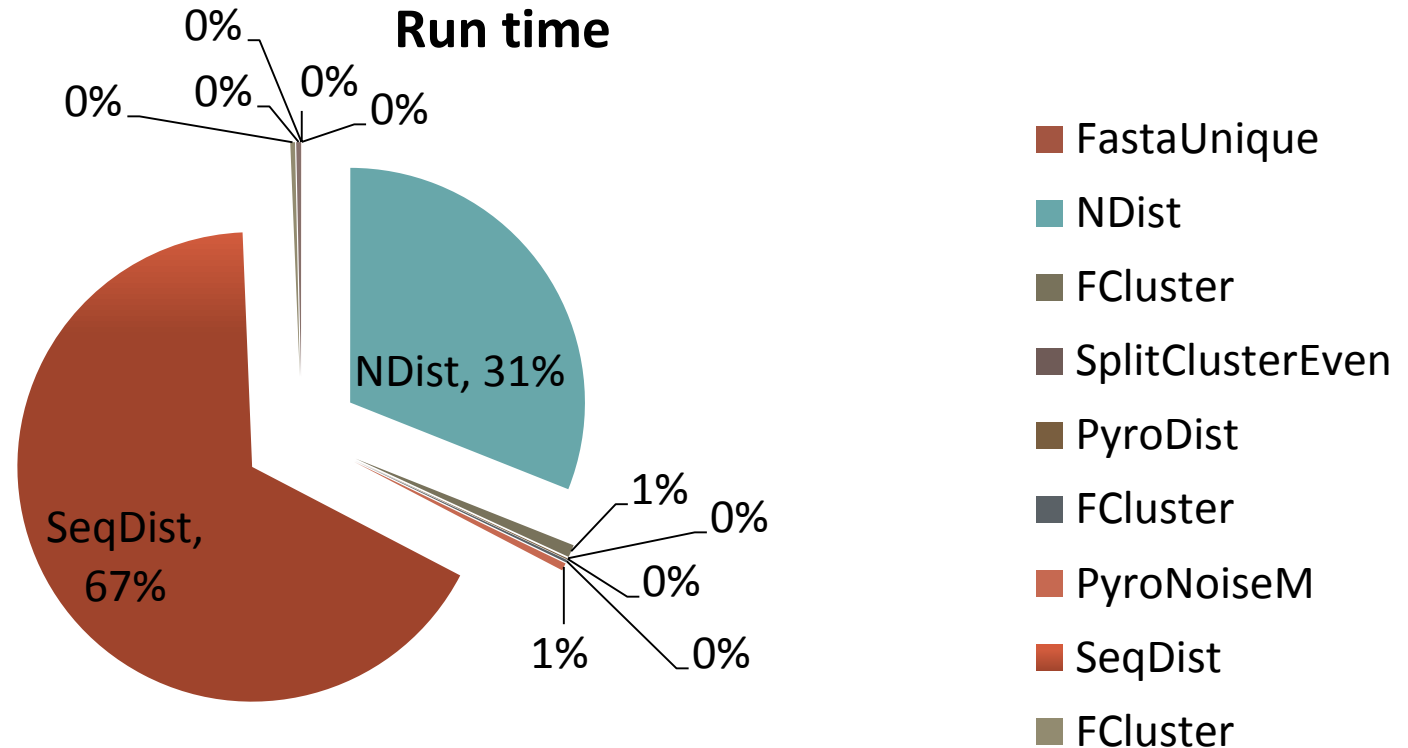
1. FastaUnique	2. NDist	3. FCluster	4. SplitClusterEven
00:00:20	18:17:32	00:30:24	00:00:00

5. PyroDist	6. FCluster	7. PyroNoiseM	8. SeqDist
00:02:51	00:06:11	00:18:22	39:20:30

9. FCluster	10. SeqNoise	11. NDist	12. Fcluster
00:11:26	00:11:31	00:00:04	00:00:00

- When parallelized by MPI, it takes **58h 59m 12s** using 8 cores.

Profiling (3/3)



- Some steps take comparably long time, so it needs to be parallelized by improved solution.
- Above target steps are dealing with Needleman-Wunsch algorithm which can be parallelized by CUDA.

Needleman-Wunsch Algorithm

- **Pairwise distance matrix**

- Time complexity = $O(n^2N^2)$
- Space complexity = $O(n^2N^2)$
 - (n : # flowgram, N : read length)

- **Dynamic programming**

- $F_{i,j} = \max(F_{i-1,j-1} + S(A_i, B_j), F_{i,j-1} + d, F_{i-1,j} + d)$
 - (F : each cell, S : score, d : gap penalty)

		A	A	G
	0	-2	-4	-6
A	-2			
A	-4			
A	-6			
C	-8			

1. Initialize Score Matrix

		A	A	G
	0	-2	-4	-6
A	-2	1	-1	
A	-4			
A	-6			
C	-8			

2. Calculate each Cell

		A	A	G
	0	-2	-4	-6
A	-2	1	-1	-3
A	-4	-1	0	-2
A	-6	-3	-2	-1
C	-8	-5	-4	-1

3. Backtracking

Needleman-Wunsch Algorithm

- **Example**

- Consider the two DNA sequences to be globally aligned are:
 - ATCG ($x = 4$, length of sequence 1)
 - TCG ($y = 3$, length of sequence 2)
- Scoring Scheme
 - Match Score = +1
 - Mismatch Score = -1
 - Gap Penalty = -1

Needleman-Wunsch Algorithm

- **Example**

- Initialization Step

- Create a matrix with $x + 1$ rows and $y + 1$ columns.
 - The 1st row and the 1st column of the score matrix are filled as **multiple of gap penalty**.

		T	C	G
	0	-1	-2	-3
A	-1			
T	-2			
C	-3			
G	-4			

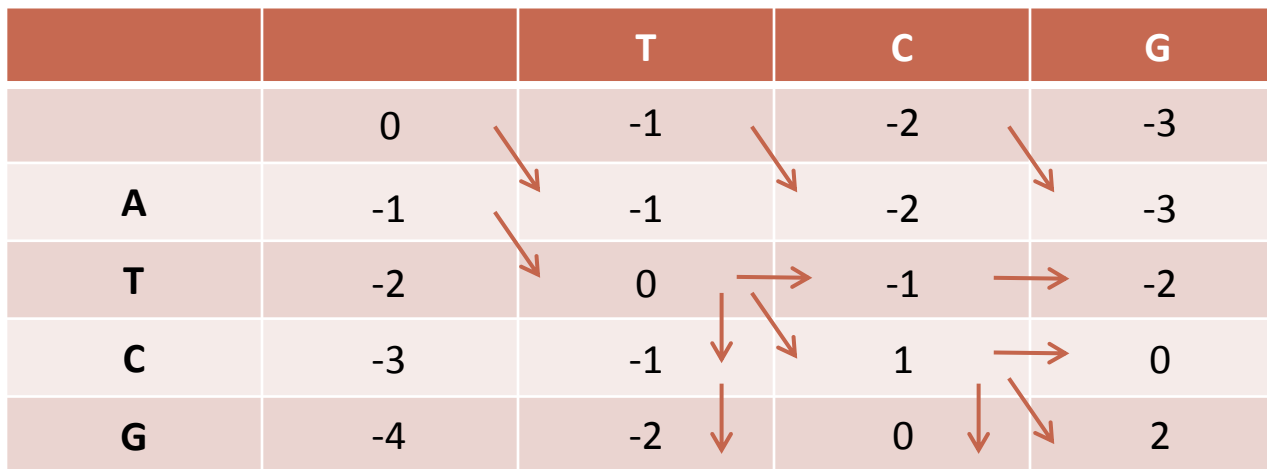
Needleman-Wunsch Algorithm

- **Example**

- Scoring Step

- The calculation for the cell $F(2,2)$:
 - Diagonal = $F(i - 1, j - 1) + S(i, j) = 0 + -1 = -1$
 - Up = $F(i - 1, j) + g = -1 + -1 = -2$
 - Left = $F(i, j - 1) + g = -1 + -1 = -2$

		T	C	G
	0	-1	-2	-3
A	-1	-1	-2	-3
T	-2	0	-1	-2
C	-3	-1	1	0
G	-4	-2	0	2



Needleman-Wunsch Algorithm

- **Example**

- **Backtracking Step**

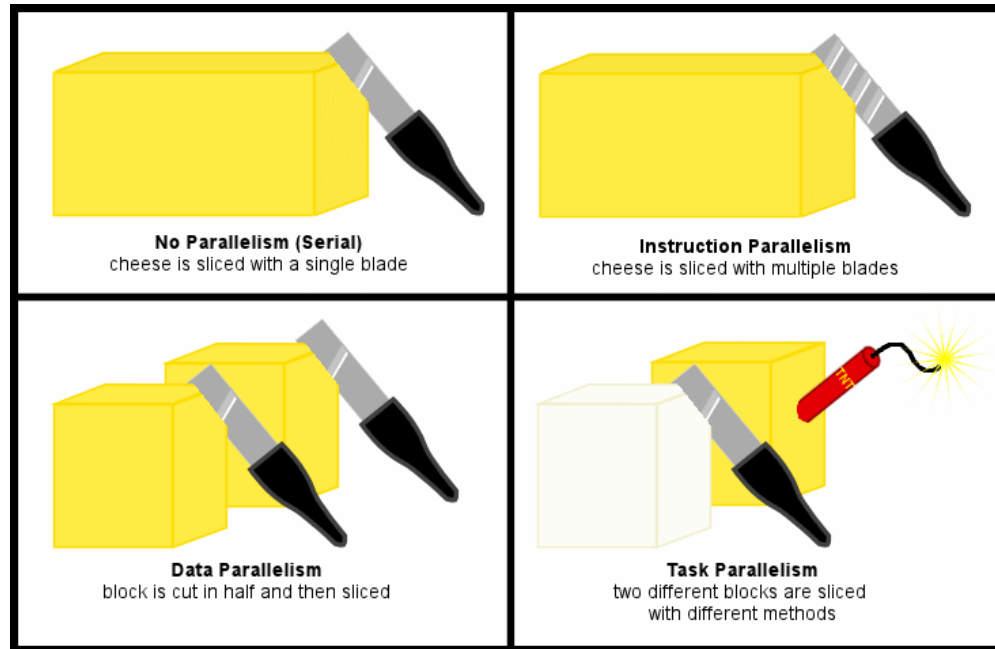
- Trace back starts from the last cell, i.e. position (x, y) in the matrix.
- Gives alignment in **reverse order**.

		T	C	G
	0	-1	-2	-3
A	-1	-1	-2	-3
T	-2	0	-1	-2
C	-3	-1	1	0
G	-4	-2	0	2

- Sequence 1: A T C G
 | | | |
- Sequence 2: - T C G

Parallelism

- **Task Parallelism (CPU)**
 - Focuses on distributing execution processes across different parallel computing nodes.
- **Data Parallelism (GPU)**
 - Focuses on distributing data across different parallel computing nodes.



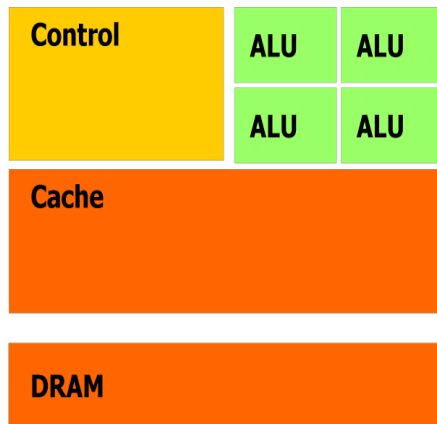
- Needleman-Wunsch algorithm can be accelerated by exploiting **data parallelism**.

- **G**eneral-**P**urpose Computation on **G**raphics **P**rocessing **U**nits
 - High-performance many-core processors
 - Can be used to accelerate a wide variety of applications

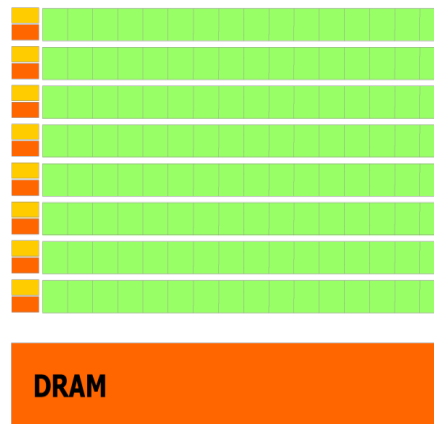


CPU vs. GPU

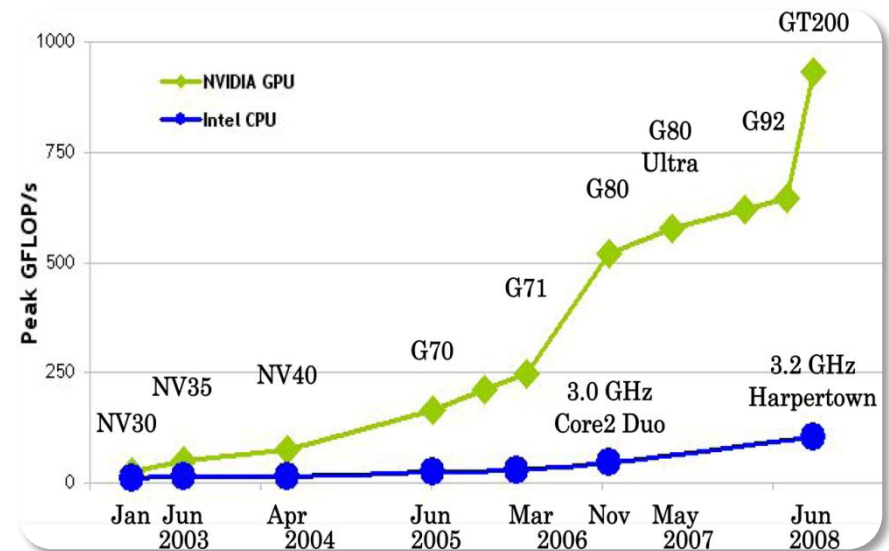
- GPU is specialized for compute-intensive, highly parallel computation.
 - No data dependency between the pixel operations.
- More transistors are devoted to data processing rather than data caching and flow control.



CPU



GPU



Source: NVIDIA, *NVIDIA CUDA C Programming Guide*.

CUDA (1/2)

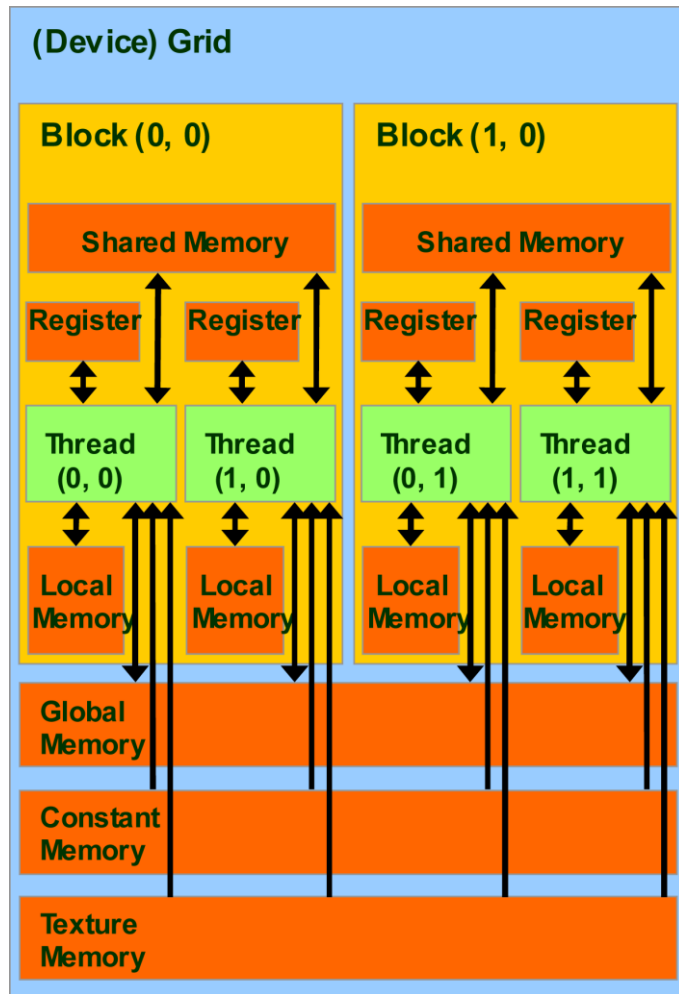
- **C**ompute **U**nified **D**evice **A**rchitecture
- Designed and developed by NVIDIA
- C programming language on GPUs
- Requires no knowledge of graphics APIs or GPU programming
- Easy to get started and to get performance benefits



- Heterogeneous programming
 - GPU is viewed as a compute device operating as a coprocessor to the main CPU(host).
- A function compiled for device is called a **kernel**.
- A **kernel** is executed by a **grid** of **thread blocks**.
- A **thread block** is a batch of **threads**.
 - Sharing data through shared memory.
 - Synchronizing their execution.

	Host (CPU)	Device (GPU)
Threading Resources	Dozens	Billions
Threads	Heavy	Light

Device Memory Hierarchy



Source: NVIDIA, NVIDIA CUDA C Programming Guide.

- Device code can :
 - R/W per-thread **register**
 - R/W per-thread **local memory**
 - R/W per-block **shared memory**
 - R/W per-grid **global memory**
 - Read only per-grid **constant** and **texture memories**
- Host code can:
 - R/W per-grid **global, constant** and **texture memories**

Features of Device Memory

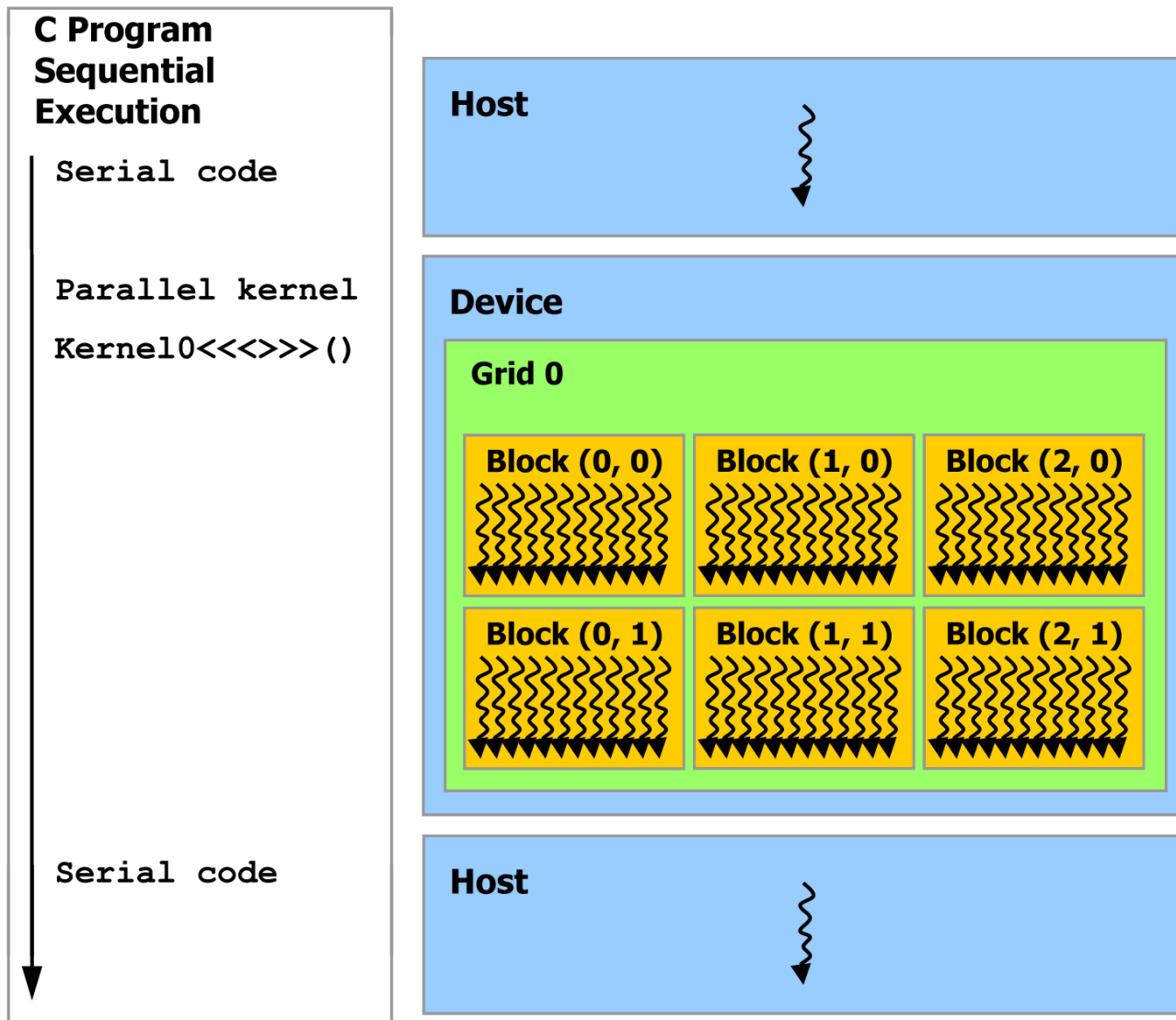
Memory	Located on chip	Cached	Access	Scope	Lifetime
Register	yes	n/a	R/W	1 thread	Thread
Local	no	no/yes	R/W	1 thread	Thread
Shared	yes	n/a	R/W	Block	Block
Global	no	no/yes	R/W	Program	Host allocation
Constant	no	yes	R	Program	Host allocation
Texture	no	yes	R	Program	Host allocation

- Caching depends on compute capability.
- **Minimize data transfer between the host and the device.**

GTX 280	GPU ↔ Device mem	Host mem ↔ Device mem
Peak bandwidth	141 GBps	8 GBps

Source: NVIDIA, **NVIDIA CUDA C Programming Guide**.

CUDA Programming Model



Source: NVIDIA, NVIDIA CUDA C Programming Guide.

CUDA Code (1/2)

```
#include "cuda.h"
#include <stdio.h>

const int BLOCK_SIZE = 16;

__global__ void VecAdd(float* A, float* B, float* C);

int main()
{
    float *h_A, *h_B; // m-by-n matrix for host
    float *d_A, *d_B; // n-by-m matrix for kernels

    // Initialize input vectors

    // Allocate vectors in device memory
    cudaMalloc((void**)&d_A, sizeof(float)*m*n); // m-by-n
    cudaMalloc((void**)&d_B, sizeof(float)*m*n); // m-by-n

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, sizeof(float)*m*n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, sizeof(float)*m*n, cudaMemcpyHostToDevice);

    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(1+m/dimBlock.x, 1+n/dimBlock.y);
    VecAdd<<< dimGrid, dimBlock >>>>(d_A, d_B, d_C); // Invoke kernel

    // Copy result from device memory to host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C); // Free device memory
}
```

Source: NVIDIA, NVIDIA CUDA C Programming Guide.



CUDA Code (2/2)

- **Kernel Function**

- Executed on the GPU, “Main function of a thread in the device”
- Single program multiple data (SPMD)

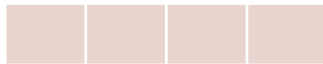
```
// Function definition
void VecAdd(float* A, float* B, float* C, int N)
{
    for(int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

**CPU
Program**

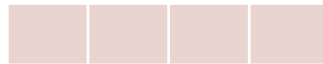
```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int idx = blockDim.x*blockIdx.x+threadIdx.x;
    C[i] = A[i] + B[i];
}
```

**CUDA
Program**

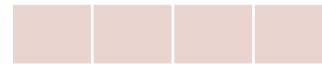
- Let's assume $N = 16$, $\text{blockDim} = 4$ (# of threads per block)



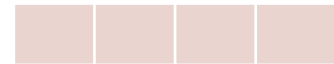
$\text{blockIdx.x} = 0$
 $\text{blockDim.x} = 4$
 $\text{threadIdx.x} = 0, 1, 2, 3$
 $i = 0, 1, 2, 3$



$\text{blockIdx.x} = 1$
 $\text{blockDim.x} = 4$
 $\text{threadIdx.x} = 0, 1, 2, 3$
 $i = 4, 5, 6, 7$



$\text{blockIdx.x} = 2$
 $\text{blockDim.x} = 4$
 $\text{threadIdx.x} = 0, 1, 2, 3$
 $i = 8, 9, 10, 11$

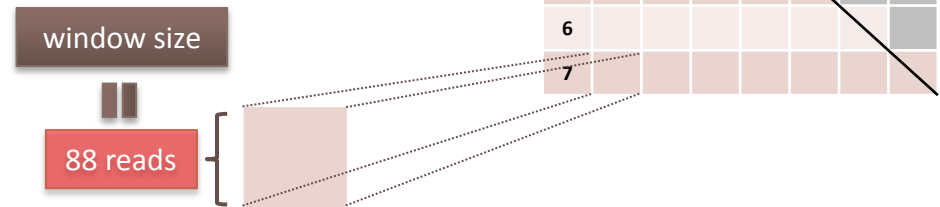


$\text{blockIdx.x} = 3$
 $\text{blockDim.x} = 4$
 $\text{threadIdx.x} = 0, 1, 2, 3$
 $i = 12, 13, 14, 15$

Parallelize AmpliconNoise (1/3)

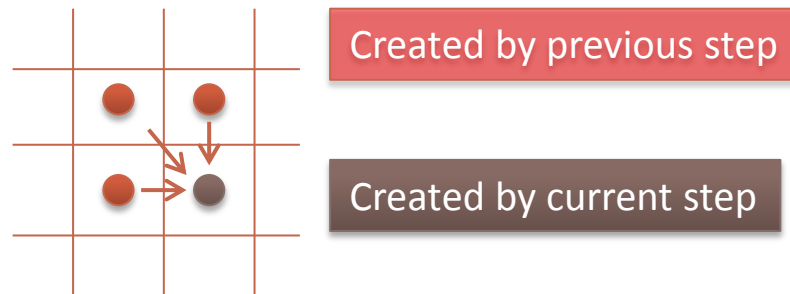
- **Divide Score Matrix**

- Device memory is limited, we divide score matrix.
- The size of **window** is determined by global memory.



- **Determine Where To Store Data**

- In each step, 4 data are needed during Needleman-Wunsch algorithm.
- Performance depends on using appropriate memory.



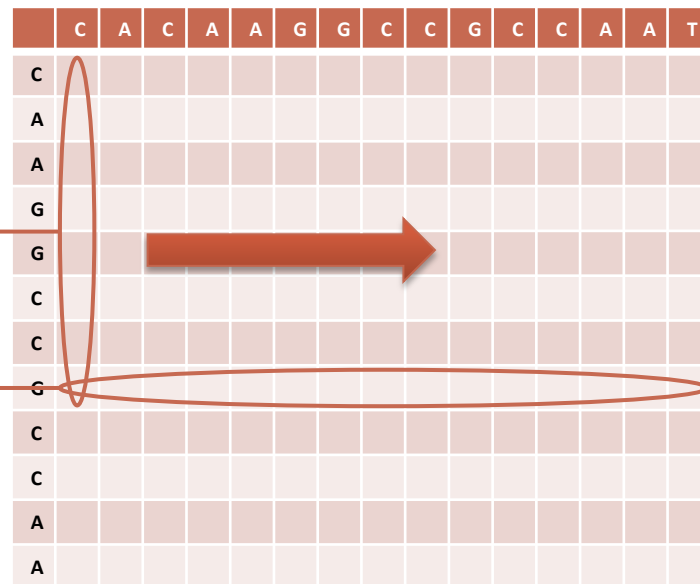
Parallelize AmpliconNoise (2/3)

- **Using Global Memory**

- Stores previous row-wise scores.
- Stored previous scores are used by a **thread** during next iteration.

Stored in shared memory

Stored in global memory



- **Using Shared Memory**

- Stores column-wise sequences and current scores.
- Stored sequences are shared by a **block**.
- Stored current scores are used by a **thread**.

Parallelize AmpliconNoise (2/3)

- **Using Global Memory**

- Stores previous row-wise scores.
- Stored previous scores are used by a **thread** during next iteration.

Stored in shared memory

	C	A	C	A	A	G	G	C	C	G	C	C	A	A	T
C	0	9	0	9	0										
A	3	1	3	1	3										
A	3	2	3	2	3										
G	0	1	0	1											
G	1	8	1	8											
C	0	9	0	9											
C	0	0	0	0											
G	1	9	1	9											
C															
C															
A															
A															

- **Using Shared Memory**

- Stores column-wise sequences and current scores.
- Stored sequences are shared by a **block**.
- Stored current scores are used by a **thread**.

Parallelize AmpliconNoise (2/3)

- **Using Global Memory**

- Stores previous row-wise scores.
- Stored previous scores are used by a **thread** during next iteration.

Stored in global memory

	C	A	C	A	A	G	G	C	C	G	C	C	A	A	T
C	0	9	0	9	0	9	0	9	0	9	0	9	0	9	0
A	3	1	3	1	3	1	3	1	3	1	3	1	3	1	3
A	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3
G	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
G	1	8	1	8	1	8	1	8	1	8	1	8	1	8	1
C	0	9	0	9	0	9	0	9	0	9	0	9	0	9	0
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	1	9	1	9	1	9	1	9	1	9	1	9	1	9	1
C															
C															
A															
A															

- **Using Shared Memory**

- Stores column-wise sequences and current scores.
- Stored sequences are shared by a **block**.
- Stored current scores are used by a **thread**.

Parallelize AmpliconNoise (2/3)

- **Using Global Memory**

- Stores previous row-wise scores.
- Stored previous scores are used by a **thread** during next iteration.

Stored in global memory

	C	A	C	A	A	G	G	C	C	G	C	C	A	A	T
C															
A															
A															
G															
G															
C															
C															
G	1	9	1	9	1	9	1	9	1	9	1	9	1	9	1
C	0														
C	3														
A	3														
A															

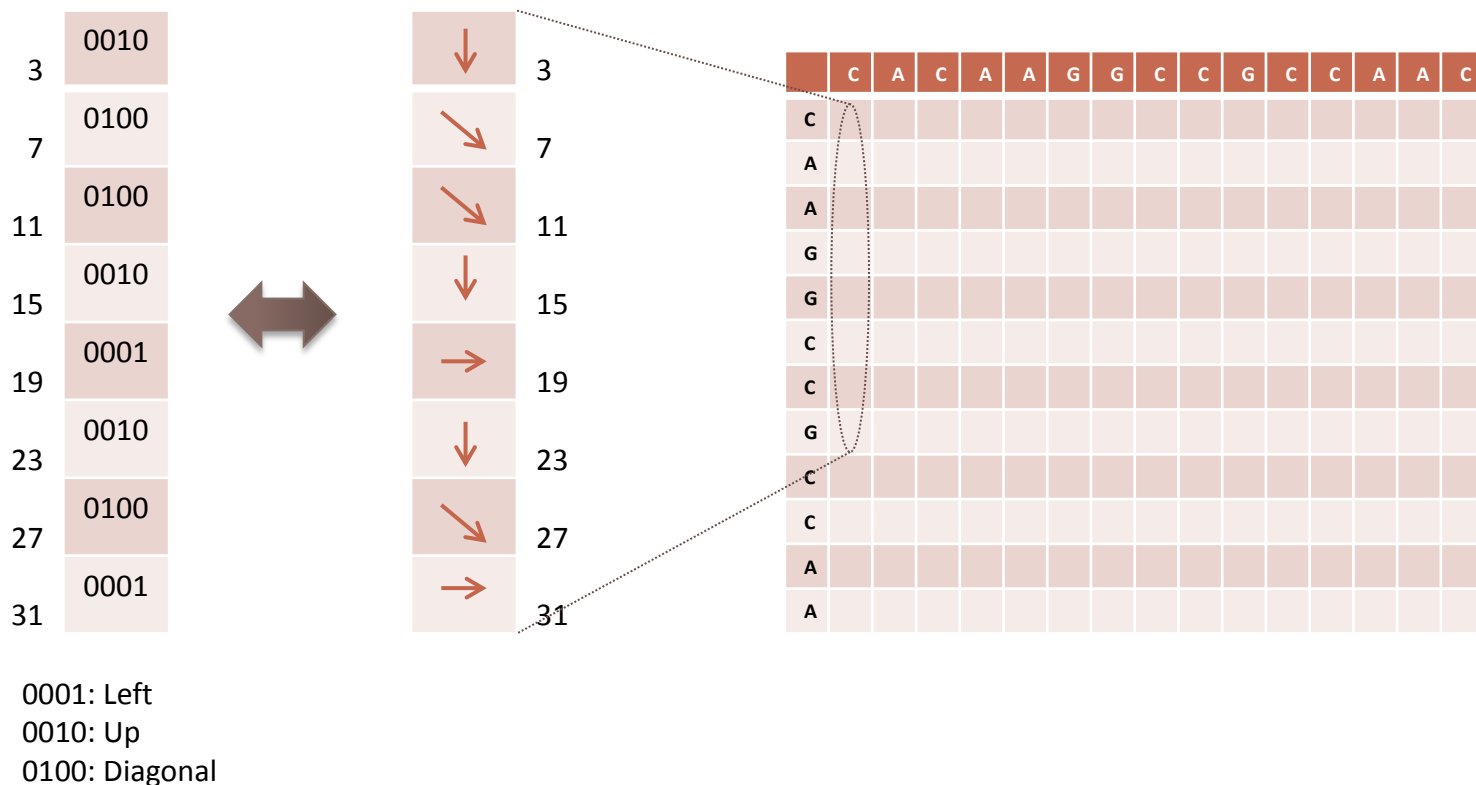
- **Using Shared Memory**

- Stores column-wise sequences and current scores.
- Stored sequences are shared by a **block**.
- Stored current scores are used by a **thread**.

Parallelize AmpliconNoise (3/3)

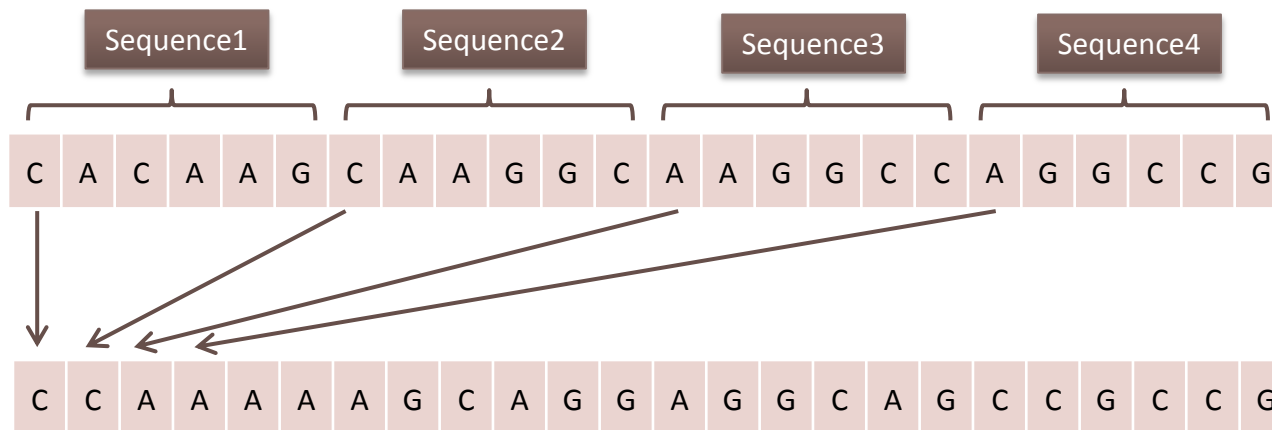
- **Compressing Backtracking Matrix**

- Space complexity = $O(n^2 N^2)$
- To reduce memory usage, we compress it by every 8 cells.
- Each cell takes up 4bit, so 8 cells can be compressed to 32bit.



Optimization

- **Using Coalesced Access Pattern**



- **Reducing Branch Condition**

- Any flow control instruction can significantly affect the instruction throughput by causing threads of the same warp to diverge.

- **Considering Occupancy**

- Determine the best size of threads per block by calculating occupancy.

Results

Results

- The tests were run on the following hardware:
 - CPU: 2x Intel Xeon X5650 @ 2.67Ghz
 - RAM: 64GB
 - GPU: 4x NVIDIA GeForce GTX 580 with 3GB of RAM
 - OS: Ubuntu Server 10.04.3 LTS 64bit
- The tests were run on the following data set[1]:
 - Generated by pyrosequencing a mixture of 91 full length 16S rRNA clones obtained from an Arctic soil sample.
 - # of Reads: 62,873



Results

- Before CUDA Parallelization (8-thread MPI version, hh:mm:ss)

1. FastaUnique	2. NDist	3. FCluster	4. SplitClusterEven
00:00:20	18:17:32	00:30:24	00:00:00

5. PyroDist	6. FCluster	7. PyroNoiseM	8. SeqDist
00:02:51	00:06:11	00:18:22	39:20:30

9. FCluster	10. SeqNoise	11. NDist	12. Fcluster
00:11:26	00:11:31	00:00:04	00:00:00

Results

- Using 1 GPU

1. FastaUnique	2. NDist	3. FCluster	4. SplitClusterEven
00:00:07	01:24:07	00:21:09	00:00:00

5. PyroDist	6. FCluster	7. PyroNoiseM	8. SeqDist
00:01:43	00:04:37	00:12:13	04:04:31

9. FCluster	10. SeqNoise	11. NDist	12. FCluster
00:08:02	00:03:37	00:00:03	00:00:00

- We parallelized Needleman-Wunsch algorithm part by using CUDA.
- When parallelized by CUDA, it takes **6h 19m 44s** using 1 GPU.

Results

- Using 4 GPUs

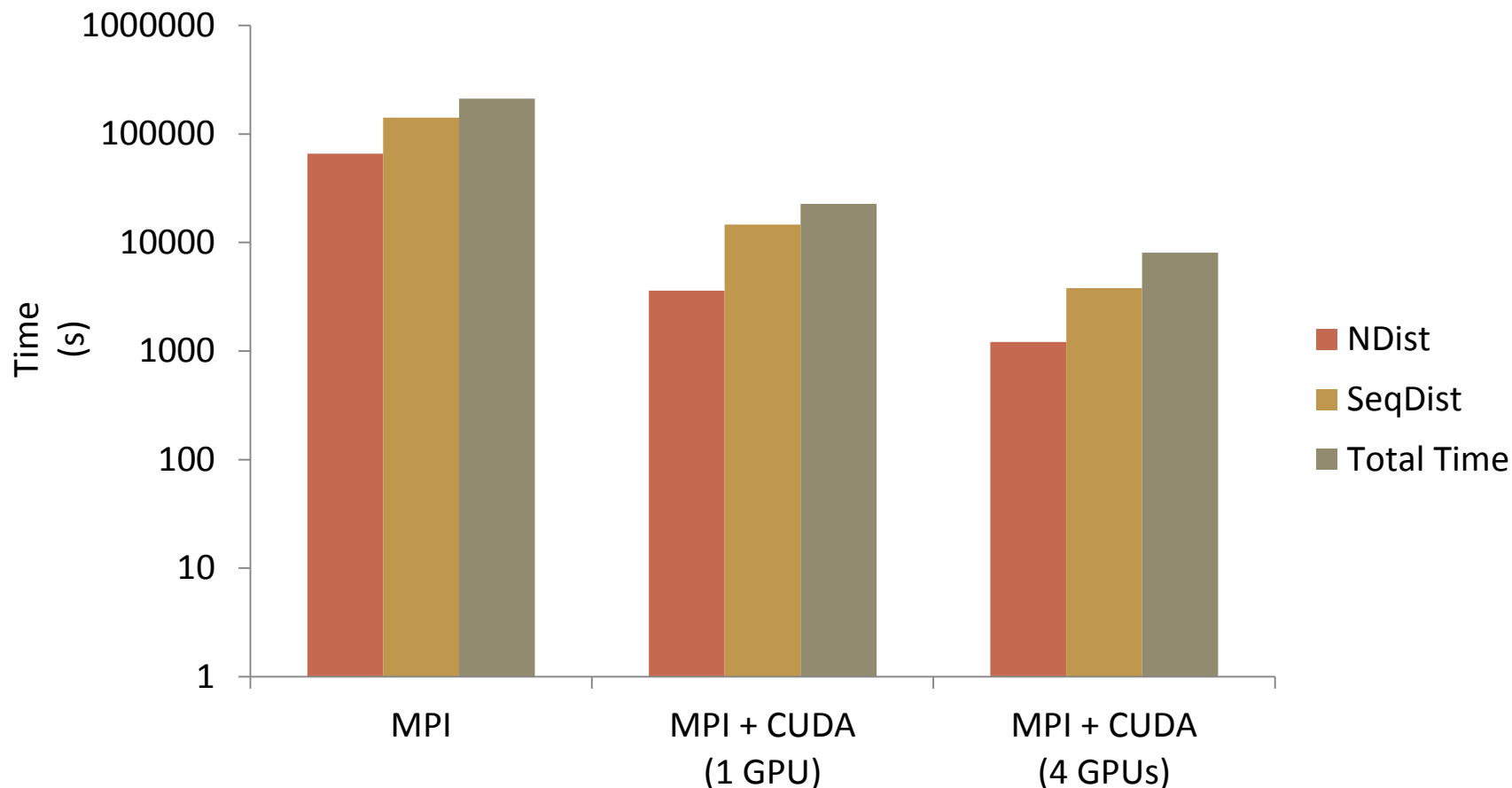
1. FastaUnique	2. NDist	3. FCluster	4. SplitClusterEven
00:00:08	00:20:15	00:21:33	00:00:00

5. PyroDist	6. FCluster	7. PyroNoiseM	8. SeqDist
00:01:20	00:04:18	00:12:13	01:03:14

9. FCluster	10. SeqNoise	11. NDist	12. FCluster
00:07:57	00:03:32	00:00:02	00:00:00

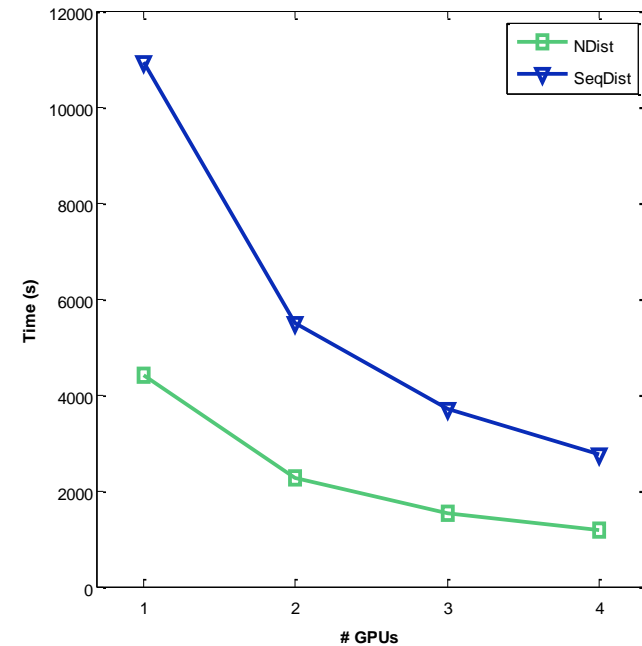
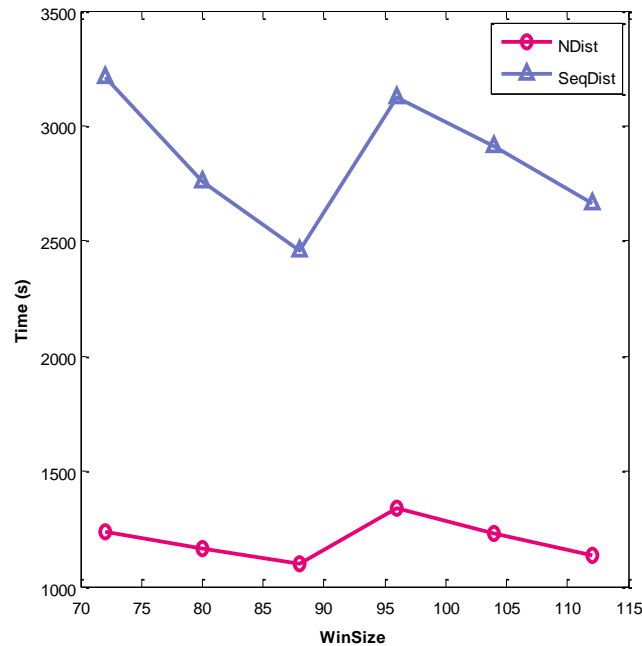
- When parallelized by CUDA, it takes **2h 14m 37s** using 4 GPU.

Results



- When using 1 GPU, **9.34 times** faster than MPI using 8 cores.
- When using 4 GPUs, **26.29 times** faster than MPI using 8 cores.

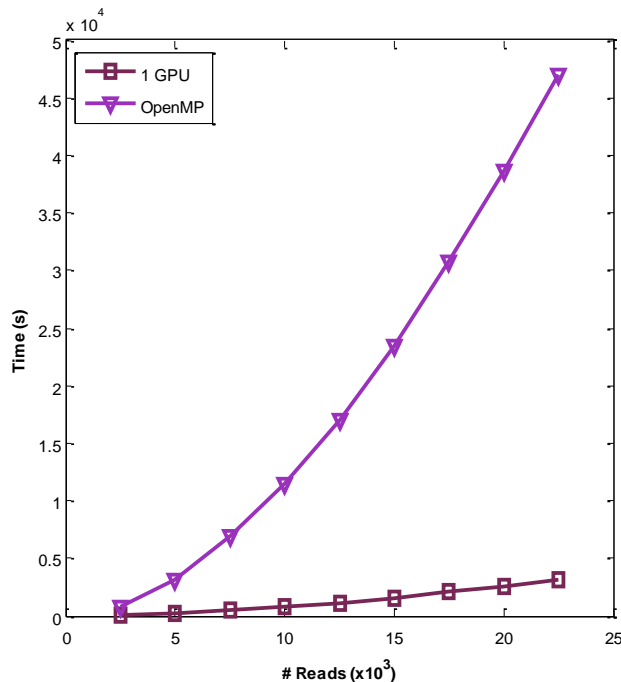
Results



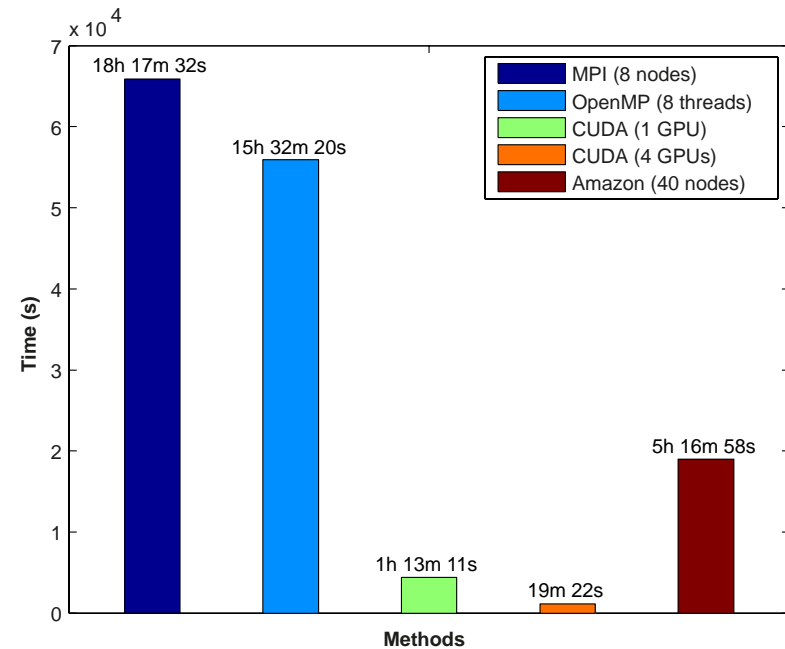
Data: A mixture of 91 full length 16S rRNA clones obtained from an Arctic soil sample[1].

- We concluded that when the window size is set to 88, the performance is best.
 - Speed-up is limited by the fixed size GPU register.
 - Inappropriate selection of window size can reduce the GPU occupancy.
- As # GPUs increasing, speed-up is proportional to that.

Results



Data: A mixture of 91 full length 16S rRNA clones obtained from an Arctic soil sample[1].



- Then we compare the performance between MPI, OpenMP, CUDA and Cloud.
 - OpenMP gives better performance than MPI since MPI has more communication latency than that of OpenMP.
 - Cloud is a easily scalable method than MPI and OpenMp.
 - But, CUDA is an inexpensive and effective method in dealing with data parallelism.

Discussion

- If the length of reads is getting longer, we have to adjust the window size.
 - The window size depends on the GPU RAM.
- The # reads does not matter in calculating distance due to the preprocessing step, but it does in the clustering step.
- Compare the methods to parallelize the algorithm.
 - MPI
 - OpenMP
 - CUDA
 - Cloud



Summary

- Pyrosequenced amplicons need to be denoised
 - Flowgram signal intensity distributions
 - PCR per base error probabilities
 - Chimeras
- The denoising algorithm can be parallelized by exploiting data parallelism
 - Alternatives: MPI, OpenMP, CUDA and cloud computing
 - MPI and cloud: relatively high communication latency
 - MPI and OpenMP: expensive to scale up
 - CUDA: relatively inexpensive and effective
 - more than 25x speed-up wrt 8-thread MPI

References

- [1] Christopher Quince, Anders Lanzen, Russell J Davenport and Peter J Turnbaugh, “Removing Noise From Pyrosequenced Amplicons”, BMC Bioinformatics, 2011.
- [2] Christopher Quince, Anders Lanzen. Thomas P Curtis, Russell J Davenport, Neil Hall, Ian M Head, L Fiona Read and William T Sloan, “Accurate determination of microbial diversity from 454 pyrosequencing data”, Nature Methods, 2009.
- [3] Jacek Blazewicz, Wojciech Frohmberg, Michal Kierzynka, Erwin Pesch and Pawel Wojciechowski, “Protein alignment algorithms with an efficient backtracking routine on multiple GPUs”, BMC Bioinformatics, 2011.
- [4] Chris Fraley and Adrian E. Raftery, “How Many Clusters? Which Clustering Method? Answers Via Model-Based Cluster Analysis”, The Computer Journal, 1998.

Any Questions?

