2022 APAC HPC-AI

Team NTHU-1 Presentation

Hao-Lung, Hsiao   Hsin-Ping, Peng
Pin-Syuan, Lee   Chun-Mu, Weng
Hsin-Cheng, Tu   Jing-Yu, Yang

Dept. of Computer Science, Nat'l Tsing Hua U.

October 24, 2022

2022-10-24

2022 APAC HPC-AI

2022 APAC HPC-AI
Team NTHU-1 Presentation

Hao-Lung, Hsiao   Hsin-Ping, Peng
Pin-Syuan, Lee   Chun-Mu, Weng
Hsin-Cheng, Tu   Jing-Yu, Yang

Dept. of Computer Science, Nat'l Tsing Hua U.

October 24, 2022

2022 APAC HPC-AI

1 High Performance Computing with QUANTUM ESPRESSO
- Parameters
- Single Node
- Multiple Nodes
- Additional Supplement

2 Communications Performance with UCX
- Introduction
- Optimized Configurations
- Conclusion
- Running on DGX-A100

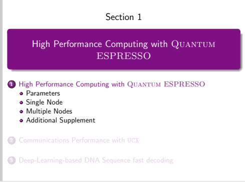
3 Deep-Learning-based DNA Sequence fast decoding
- Why do we choose this model?
- Model structure
- Tuning
  - Batch size
- Result

Section 1

High Performance Computing with QUANTUM ESPRESSO

2022-10-24

2022 APAC HPC-AI
└─High Performance Computing with QUANTUM ESPRESSO



Section 1

High Performance Computing with QUANTUM ESPRESSO

1. High Performance Computing with QUANTUM ESPRESSO
   - Parameters
   - Single Node
   - Multiple Nodes
   - Additional Supplement
2. Communications Performance with UCX
3. Deep-Learning-based DNA Sequence fast decoding

Parameters

-np, -diag, OMP_NUM_THREADS

Number of Processes (-np)

- Each node has 48 processes
- Cannot greater than 1536, must be an integer in $[1, 48]$ or $48x | 1 \leq x \leq 32$
- Has effects on `-npool` & diagonalization

Diagonalization (-ndiag)

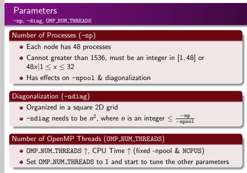
- Organized in a square 2D grid
- `-ndiag` needs to be n^2, where n is an integer $\leq \frac{-\texttt{np}}{-\texttt{npool}}$

Number of OpenMP Threads (OMP_NUM_THREADS)

- OMP_NUM_THREADS $\uparrow$, CPU Time $\uparrow$ (fixed -npool & NCPUS)
- Set OMP_NUM_THREADS to 1 and start to tune the other parameters

2022-10-24

2022 APAC HPC-AI
└High Performance Computing with Quantum ESPRESSO
 └Parameters
  └Parameters

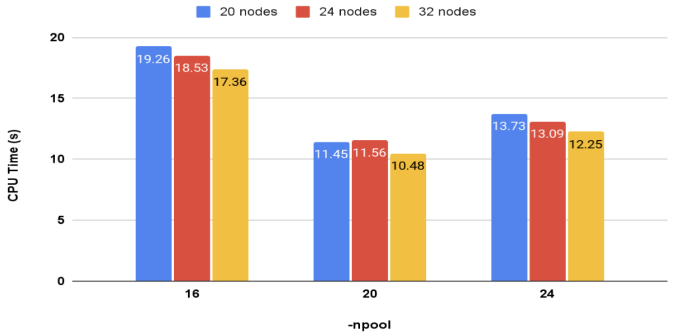

Parameters
-np, -diag, OMP_NUM_THREADS
Number of Processes (-np)
- Each node has 48 processes
- Cannot greater than 1536, must be an integer in $[1, 48]$ or $48x | 1 \leq x \leq 32$
- Has effects on -npool & diagonalization
Diagonalization (-ndiag)
- Organized in a square 2D grid
- -ndiag needs to be n^2, where n is an integer $\leq \frac{-\texttt{np}}{-\texttt{npool}}$
Number of OpenMP Threads (OMP_NUM_THREADS)
- OMP_NUM_THREADS $\uparrow$, CPU Time $\uparrow$ (fixed -npool & NCPUS)
- Set OMP_NUM_THREADS to 1 and start to tune the other parameters

Parameters

`-npool`

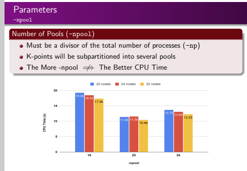
Number of Pools (`-npool`)

- Must be a divisor of the total number of processes (`-np`)
- K-points will be subpartitioned into several pools
- The More -npool $\not\Rightarrow$ The Better CPU Time



2022-10-24

2022 APAC HPC-AI
└High Performance Computing with QUANTUM ESPRESSO
 └Parameters
  └Parameters



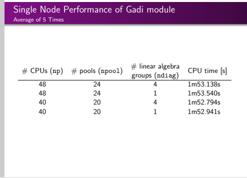
Single Node Performance of Gadi module

Average of 5 Times

# CPUs (np)	# pools (npool)	# linear algebra groups (ndiag)	CPU time [s]
48	24	4	1m53.138s
48	24	1	1m53.540s
40	20	4	1m52.794s
40	20	1	1m52.941s

2022-10-24

2022 APAC HPC-AI
└High Performance Computing with QUANTUM ESPRESSO
└Single Node
└Single Node Performance of Gadi module


Single Node Performance of Gadi module
Average of 5 Times

# CPUs (np)	# pools (npool)	# linear algebra groups (ndiag)	CPU time [s]
48	24	4	1m53.138s
48	24	1	1m53.540s
40	20	4	1m52.794s
40	20	1	1m52.941s

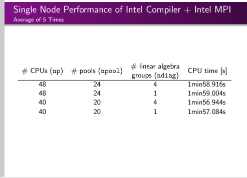
Single Node Performance of Intel Compiler + Intel MPI

Average of 5 Times

# CPUs (np)	# pools (npool)	# linear algebra groups (ndiag)	CPU time [s]
48	24	4	1min58.916s
48	24	1	1min59.004s
40	20	4	1min56.944s
40	20	1	1min57.084s

2022-10-24

2022 APAC HPC-AI
└─High Performance Computing with QUANTUM ESPRESSO
  └─Single Node
    └─Single Node Performance of Intel Compiler + Intel MPI

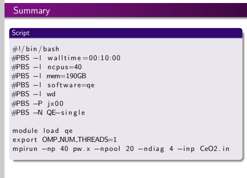

Single Node Performance of Intel Compiler + Intel MPI
Average of 5 Times

# CPUs (np)	# pools (npool)	# linear algebra groups (ndiag)	CPU time [s]
48	24	4	1min58.916s
48	24	1	1min59.004s
40	20	4	1min56.944s
40	20	1	1min57.084s

Summary

Script

```
#!/bin/bash
#PBS -l walltime=00:10:00
#PBS -l ncpus=40
#PBS -l mem=190GB
#PBS -l software=qe
#PBS -l wd
#PBS -P jx00
#PBS -N QE-single

module load qe
export OMP_NUM_THREADS=1
mpirun -np 40 pw.x -npool 20 -ndiag 4 -inp CeO2.in
```

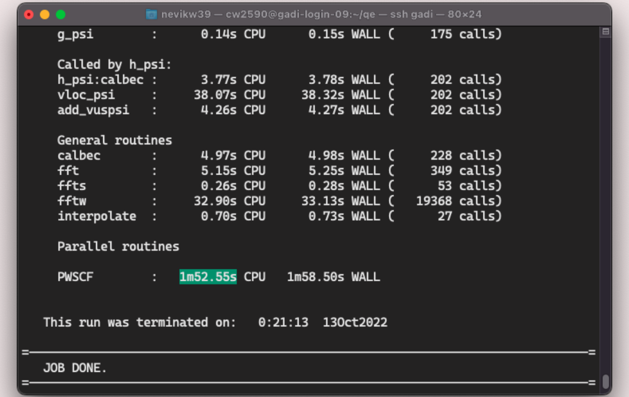

Summary

Script

```
#!/bin/bash
#PBS -l walltime=00:10:00
#PBS -l ncpus=40
#PBS -l mem=190GB
#PBS -l software=qe
#PBS -l wd
#PBS -P jx00
#PBS -N QE-single

module load qe
export OMP_NUM_THREADS=1
mpirun -np 40 pw.x -npool 20 -ndiag 4 -inp CeO2.in
```

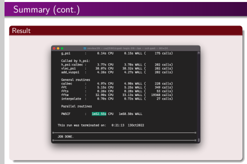
Summary (cont.)

Result



nevikw39 — cw2590@gadi-login-09:~/qe — ssh gadi — 80×24

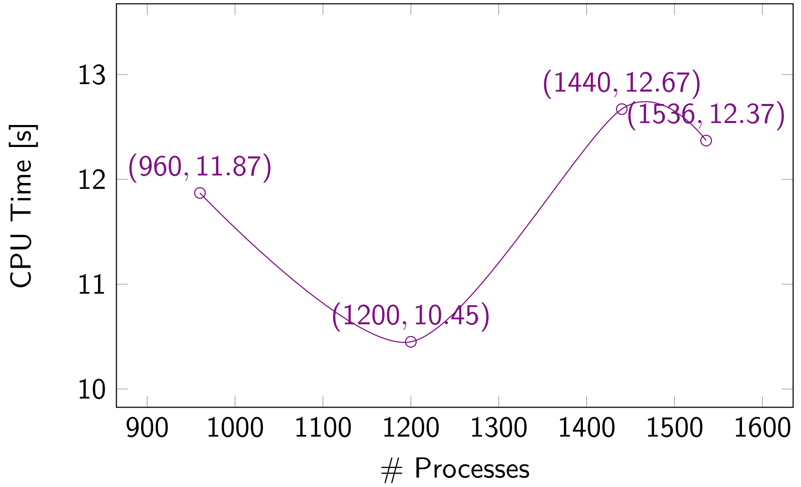
```
     g_psi        :      0.14s CPU      0.15s WALL (     175 calls)

     Called by h_psi:
     h_psi:calbec :      3.77s CPU      3.78s WALL (     202 calls)
     vloc_psi     :     38.07s CPU     38.32s WALL (     202 calls)
     add_vuspsi   :      4.26s CPU      4.27s WALL (     202 calls)

     General routines
     calbec       :      4.97s CPU      4.98s WALL (     228 calls)
     fft          :      5.15s CPU      5.25s WALL (     349 calls)
     ffts         :      0.26s CPU      0.28s WALL (      53 calls)
     fftw         :     32.90s CPU     33.13s WALL (   19368 calls)
     interpolate  :      0.70s CPU      0.73s WALL (      27 calls)

     Parallel routines

     PWSCF        :   1m52.55s CPU   1m58.50s WALL


   This run was terminated on:   0:21:13  13Oct2022

=------------------------------------------------------------------------------=
   JOB DONE.
=------------------------------------------------------------------------------=
```



CPU Time vs. # Processes

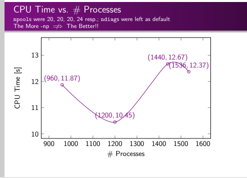
`npools` were 20, 20, 20, 24 resp.; `ndiags` were left as default

The More -np $\not\Rightarrow$ The Better!!

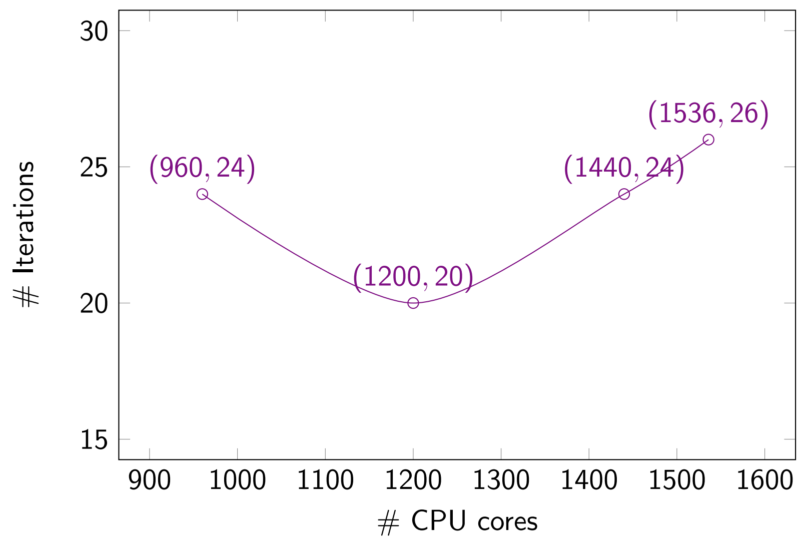


2022-10-24

2022 APAC HPC-AI
└High Performance Computing with QUANTUM ESPRESSO
 └Multiple Nodes
  └CPU Time vs. # Processes

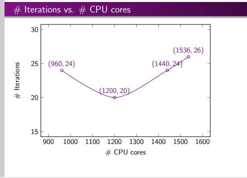


# Iterations vs. # CPU cores



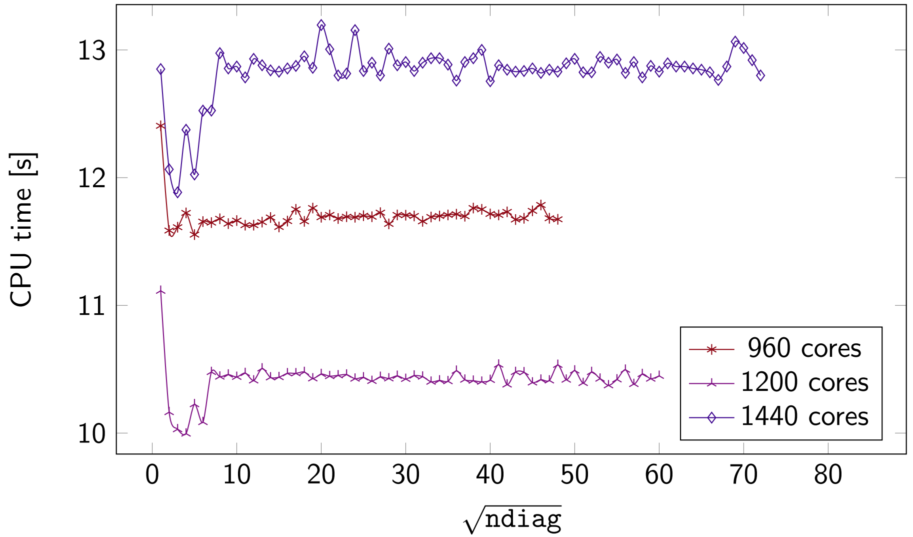
2022-10-24

2022 APAC HPC-AI
└High Performance Computing with QUANTUM ESPRESSO
└Multiple Nodes
└# Iterations vs. # CPU cores



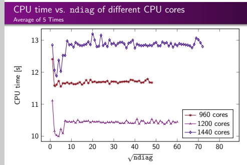
CPU time vs. ndiag of different CPU cores

Average of 5 Times



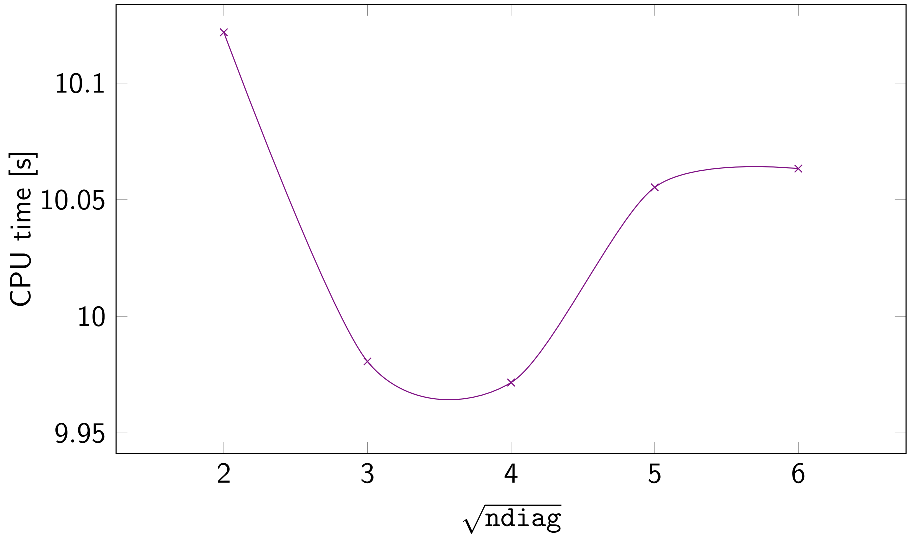
2022-10-24

2022 APAC HPC-AI
└High Performance Computing with Quantum ESPRESSO
 └Multiple Nodes
  └CPU time vs. ndiag of different CPU cores



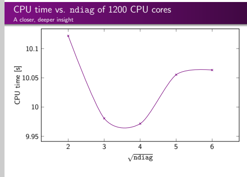
CPU time vs. ndiag of 1200 CPU cores

A closer, deeper insight



2022-10-24

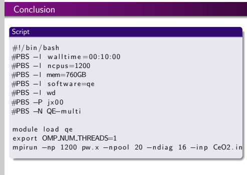
2022 APAC HPC-AI
└High Performance Computing with QUANTUM ESPRESSO
 └Multiple Nodes
  └CPU time vs. ndiag of 1200 CPU cores



Conclusion

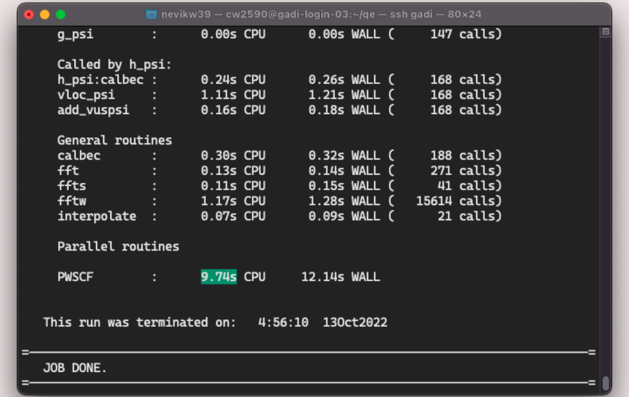
Script

```
#!/bin/bash
#PBS -l walltime=00:10:00
#PBS -l ncpus=1200
#PBS -l mem=760GB
#PBS -l software=qe
#PBS -l wd
#PBS -P jx00
#PBS -N QE-multi

module load qe
export OMP_NUM_THREADS=1
mpirun -np 1200 pw.x -npool 20 -ndiag 16 -inp CeO2.in
```


Conclusion
Script
#!/bin/bash
#PBS -l walltime=00:10:00
#PBS -l ncpus=1200
#PBS -l mem=760GB
#PBS -l software=qe
#PBS -l wd
#PBS -P jx00
#PBS -N QE-multi
module load qe
export OMP_NUM_THREADS=1
mpirun -np 1200 pw.x -npool 20 -ndiag 16 -inp CeO2.in

Conclusion (cont.)

Result

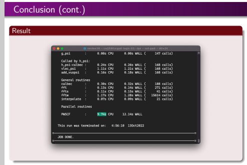


nevikw39 — cw2590@gadi-login-03:~/qe — ssh gadi — 80×24

```
     g_psi        :      0.00s CPU      0.00s WALL (     147 calls)

     Called by h_psi:
     h_psi:calbec :      0.24s CPU      0.26s WALL (     168 calls)
     vloc_psi     :      1.11s CPU      1.21s WALL (     168 calls)
     add_vuspsi   :      0.16s CPU      0.18s WALL (     168 calls)

     General routines
     calbec       :      0.30s CPU      0.32s WALL (     188 calls)
     fft          :      0.13s CPU      0.14s WALL (     271 calls)
     ffts         :      0.11s CPU      0.15s WALL (      41 calls)
     fftw         :      1.17s CPU      1.28s WALL (   15614 calls)
     interpolate  :      0.07s CPU      0.09s WALL (      21 calls)

     Parallel routines

     PWSCF        :      9.74s CPU     12.14s WALL


   This run was terminated on:   4:56:10  13Oct2022

=------------------------------------------------------------------------------=
   JOB DONE.
=------------------------------------------------------------------------------=
```

2022-10-24

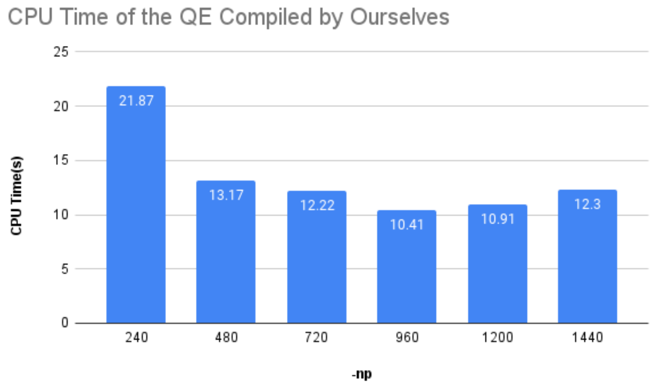
2022 APAC HPC-AI
└High Performance Computing with Quantum ESPRESSO
  └Multiple Nodes
    └Conclusion (cont.)



Additional Supplement

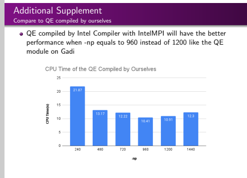
Compare to QE compiled by ourselves

- QE compiled by Intel Compiler with IntelMPI will have the better performance when -np equals to 960 instead of 1200 like the QE module on Gadi



2022-10-24

2022 APAC HPC-AI
└High Performance Computing with QUANTUM ESPRESSO
  └Additional Supplement
    └Additional Supplement

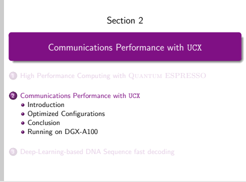

Additional Supplement
Compare to QE compiled by ourselves
QE compiled by Intel Compiler with IntelMPI will have the better performance when -np equals to 960 instead of 1200 like the QE module on Gadi

Section 2

Communications Performance with UCX

2022-10-24

2022 APAC HPC-AI
└─Communications Performance with UCX


Section 2

Communications Performance with UCX

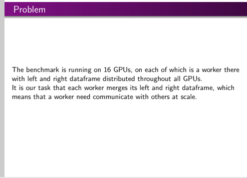
1 High Performance Computing with QUANTUM ESPRESSO

2 Communications Performance with UCX
- Introduction
- Optimized Configurations
- Conclusion
- Running on DGX-A100

3 Deep-Learning-based DNA Sequence fast decoding

Problem

The benchmark is running on 16 GPUs, on each of which is a worker there with left and right dataframe distributed throughout all GPUs.
It is our task that each worker merges its left and right dataframe, which means that a worker need communicate with others at scale.



Problem

The benchmark is running on 16 GPUs, on each of which is a worker there with left and right dataframe distributed throughout all GPUs.
It is our task that each worker merges its left and right dataframe, which means that a worker need communicate with others at scale.

`aftermath`
Our task is Communications Performance with UCX. The task is about running some python code, these codes creates a central server to synchronize workers, and each worker merges its left and right dataframe. That means the workers need communicate with each others. And our job is to improving performance for this heavy-communication work.

The given benchmark ran on 16 GPUs, on each of which is a worker there with left and right dataframe distributed throughout all GPUs.

Objectives

Bandwidth of a worker

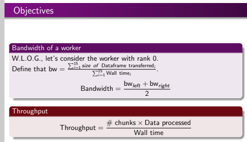
W.L.O.G., let's consider the worker with rank 0.

Define that $\text{bw} = \frac{\sum_{i=1}^{15} size\ \ of\ \ \text{Dataframe transferred}_i}{\sum_{i=1}^{15} \text{Wall time}_i}$.

$$\text{Bandwidth} = \frac{\text{bw}_{left} + \text{bw}_{right}}{2}$$

Throughput

$$\text{Throughput} = \frac{\#\ \text{chunks} \times \text{Data processed}}{\text{Wall time}}$$



nevikw39

There are two major metrics to measure the performance of the benchmark we run. We tried to discover the instinct meaning of the two by means of investigating and interpreting the codes provided.
The first one is the average bandwidth of all workers. For a particular worker, its bandwidth is the average of its left and right dataframe. For each side, it is the summation of size of dataframe transferred from and to all other workers divided by wall time.
Another one is throughput, which is the number of chunks time the total data processed by all workers divided by wall time.

We believe that there must be some relation between the two yet that throughput is more comprehensive, so we put more emphasis on it.

Baseline

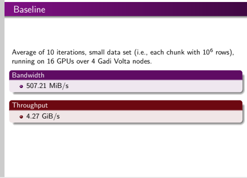
Average of 10 iterations, small data set (i.e., each chunk with 10^6 rows), running on 16 GPUs over 4 Gadi Volta nodes.

Bandwidth

- 507.21 MiB/s

Throughput

- 4.27 GiB/s


Baseline

Average of 10 iterations, small data set (i.e., each chunk with 10^6 rows), running on 16 GPUs over 4 Gadi Volta nodes.

Bandwidth
- 507.21 MiB/s

Throughput
- 4.27 GiB/s

`aftermath`

First, we run the job without modifying any code and configuration options, as our baseline. Here, the baseline ran on 16 GPUs over 4 Gadi Volta nodes, for 10 iterations, and for small data set, which means each chunk contains 10 to the power of 6 rows.

For the baseline, we got the bandwidth 507.21 megabytes per second, and the throughput 4.27 gigabyte per second.

Enable Hardware Tag Matching

Avg. of 10 iterations, small data set, Gadi Volta nodes

Config

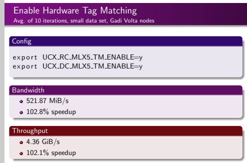
```
export UCX_RC_MLX5_TM_ENABLE=y
export UCX_DC_MLX5_TM_ENABLE=y
```

Bandwidth

- 521.87 MiB/s
- 102.8% speedup

Throughput

- 4.36 GiB/s
- 102.1% speedup



nevikw39

We managed to find several UCX options that might help. The first option is to enable the hardware tag matching for both *Reliable Connected (RC)* and *Dynamically Connected (DC)* so that these works are offload to NICs. We can see that bandwidth somewhat dropped a bit while the throughput increased by a bit. The effect of this option is not so obvious.

Enable various optimizations intended for homogeneous environment

Avg. of 10 iterations, small data set, Gadi Volta nodes

Config

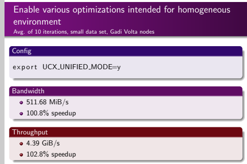
```
export UCX_UNIFIED_MODE=y
```

Bandwidth

- 511.68 MiB/s
- 100.8% speedup

Throughput

- 4.39 GiB/s
- 102.8% speedup


Enable various optimizations intended for homogeneous environment
Avg. of 10 iterations, small data set, Gadi Volta nodes
Config
export UCX_UNIFIED_MODE=y
Bandwidth
- 511.68 MiB/s
- 100.8% speedup
Throughput
- 4.39 GiB/s
- 102.8% speedup

`nevikw39`

Another option we tried is enabling the unified mode, implying that the local transport resources/devices of all entities which connect to each other are the same.

Nevertheless, this option would be conflict to the *rendezvous* scheme we would choose.

Similarly, the effect is also not so apparent.

Increase the amount of buffers added every time the receive / send memory pool grows

Avg. of 10 iterations, small data set, Gadi Volta nodes

Config

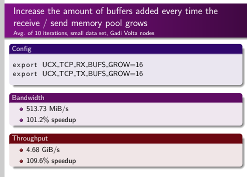
```
export UCX_TCP_RX_BUFS_GROW=16
export UCX_TCP_TX_BUFS_GROW=16
```

Bandwidth

- 513.73 MiB/s
- 101.2% speedup

Throughput

- 4.68 GiB/s
- 109.6% speedup


Increase the amount of buffers added every time the receive / send memory pool grows
Avg. of 10 iterations, small data set, Gadi Volta nodes
Config
export UCX_TCP_RX_BUFS_GROW=16
export UCX_TCP_TX_BUFS_GROW=16
Bandwidth
513.73 MiB/s
101.2% speedup
Throughput
4.68 GiB/s
109.6% speedup

`aftermath`

The two options here determine how much buffers are added every time the memory pool grows. The above one determines for the receive memory pool, while the below one determines for the send memory pool.

We have thought about that increasing the buffer grow rate may bring better performance for us. Therefore, we've tried to double the buffer grow rate. The default value of these two options are both 8, and we've modified both of them to 16. After the modification, the bandwidth increases to 513.73 MiB/s, and the throughput increases to 4.68 GiB/s, which really improves the performance.

Nonetheless, we found that this option would hardly give rise to ideal promotion in combination with others later.

Use **mutex** instead of **spinlock** for multithreading support in UCP

Avg. of 10 iterations, small data set, Gadi Volta nodes

Config

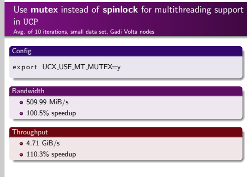
```
export UCX_USE_MT_MUTEX=y
```

Bandwidth

- 509.99 MiB/s
- 100.5% speedup

Throughput

- 4.71 GiB/s
- 110.3% speedup


Use **mutex** instead of **spinlock** for multithreading support in UCP
Avg. of 10 iterations, small data set, Gadi Volta nodes
Config
export UCX_USE_MT_MUTEX=y
Bandwidth
- 509.99 MiB/s
- 100.5% speedup
Throughput
- 4.71 GiB/s
- 110.3% speedup

`aftermath`

For this option, it determines which mechanism to use for Multithreading Support. It is set to n by default, which means not using mutex for multithreading support and using spinlock instead.

Both spinlock and mutex are common synchronization mechanism.The major difference between them is that the mechanism that mutex uses is sleep-waiting, while spinlock uses busy-waiting.

There are some pros and cons for both of them, and we have no idea which one is more efficient for the task. Hence we've tried both of them and discover that setting this option to y, which means using mutex do improve the performance. The bandwidth comes to 509.99 MiB/s, while the throughput comes to 4.71 GiB/s.

Enable UCX-Py non-blocking mode

Avg. of 10 iterations, small data set, Gadi Volta nodes

Config

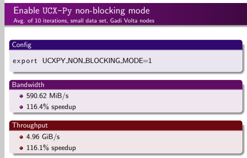
```
export UCXPY_NON_BLOCKING_MODE=1
```

Bandwidth

- 590.62 MiB/s
- 116.4% speedup

Throughput

- 4.96 GiB/s
- 116.1% speedup



nevikw39

In addition, we enabled non-blocking progress mode of UCX-Py. This time, both bandwidth and throughput are about 1.16 times the baseline.

Set *Rendezvous* protocol to use *Active Messages* scheme

Avg. of 10 iterations, small data set, Gadi Volta nodes

Config

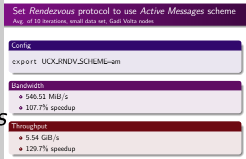
```
export UCX_RNDV_SCHEME=am
```

Bandwidth

- 546.51 MiB/s
- 107.7% speedup

Throughput

- 5.54 GiB/s
- 129.7% speedup

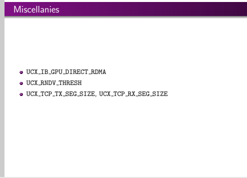

Set *Rendezvous* protocol to use *Active Messages* scheme
Avg. of 10 iterations, small data set, Gadi Volta nodes
Config
export UCX_RNDV_SCHEME=am
Bandwidth
546.51 MiB/s
107.7% speedup
Throughput
5.54 GiB/s
129.7% speedup

nevikw39

Even though this option is not documented in detail, we tried all available scheme, such as get/put zero copy, pipelined get/put zero copy, ..., finding that only active message brought a significant 30%-improvement in throughput.

Miscellanies

- UCX_IB_GPU_DIRECT_RDMA
- UCX_RNDV_THRESH
- UCX_TCP_TX_SEG_SIZE, UCX_TCP_RX_SEG_SIZE


Miscellanies
- UCX_IB_GPU_DIRECT_RDMA
- UCX_RNDV_THRESH
- UCX_TCP_TX_SEG_SIZE, UCX_TCP_RX_SEG_SIZE

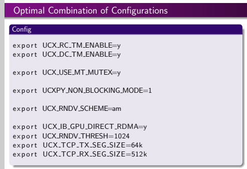
`nevikw39`

Here are some options we used to try but find no concrete conclusion, such as explicitly use GPU Direct RDMA for HCA to access GPU pages directly, reduce the threshold to switch to RNDV protocol, enlarge copy-out buffer for TCP sending and receiving.

Optimal Combination of Configurations

Config

```
export UCX_RC_TM_ENABLE=y
export UCX_DC_TM_ENABLE=y

export UCX_USE_MT_MUTEX=y

export UCXPY_NON_BLOCKING_MODE=1

export UCX_RNDV_SCHEME=am

export UCX_IB_GPU_DIRECT_RDMA=y
export UCX_RNDV_THRESH=1024
export UCX_TCP_TX_SEG_SIZE=64k
export UCX_TCP_RX_SEG_SIZE=512k
```


Optimal Combination of Configurations
Config
export UCX_RC_TM_ENABLE=y
export UCX_DC_TM_ENABLE=y
export UCX_USE_MT_MUTEX=y
export UCXPY_NON_BLOCKING_MODE=1
export UCX_RNDV_SCHEME=am
export UCX_IB_GPU_DIRECT_RDMA=y
export UCX_RNDV_THRESH=1024
export UCX_TCP_TX_SEG_SIZE=64k
export UCX_TCP_RX_SEG_SIZE=512k

`aftermath`

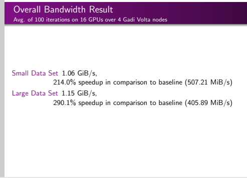
With the above attempts, we combine all configurations that would provide improvement to overall performance. The options we choose are listed here.

Overall Bandwidth Result

Avg. of 100 iterations on 16 GPUs over 4 Gadi Volta nodes

Small Data Set 1.06 GiB/s,
214.0% speedup in comparison to baseline (507.21 MiB/s)

Large Data Set 1.15 GiB/s,
290.1% speedup in comparison to baseline (405.89 MiB/s)



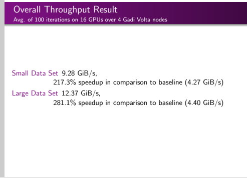
aftermath

This is our overall bandwidth result. For the small data set, the bandwidth of optimized our config is 1.06 GiB/s, which is more than twice of the baseline. And for the large data set, the bandwidth comes to 1.15 GiB/s, which is almost three times of the baseline.

Overall Throughput Result

Avg. of 100 iterations on 16 GPUs over 4 Gadi Volta nodes

Small Data Set	9.28 GiB/s, 217.3% speedup in comparison to baseline (4.27 GiB/s)
Large Data Set	12.37 GiB/s, 281.1% speedup in comparison to baseline (4.40 GiB/s)

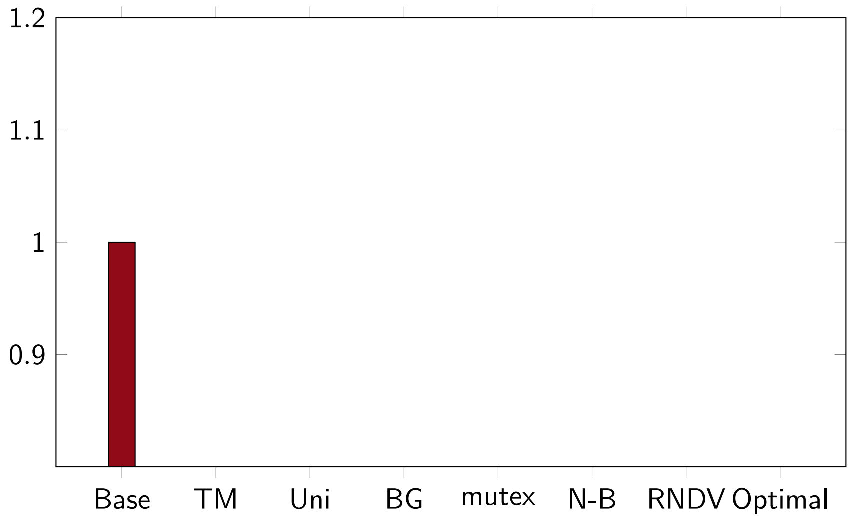

Overall Throughput Result
Avg. of 100 iterations on 16 GPUs over 4 Gadi Volta nodes
Small Data Set 9.28 GiB/s,
217.3% speedup in comparison to baseline (4.27 GiB/s)
Large Data Set 12.37 GiB/s,
281.1% speedup in comparison to baseline (4.40 GiB/s)

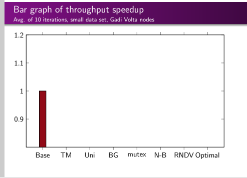
`nevikw39`

For the small data set, the throughput of optimized our config is more than twice the one of baseline. And for the large one, it's almost three times. Note that the speedup of throughput for both data set is not far from the speedup of bandwidth.

Bar graph of throughput speedup

Avg. of 10 iterations, small data set, Gadi Volta nodes



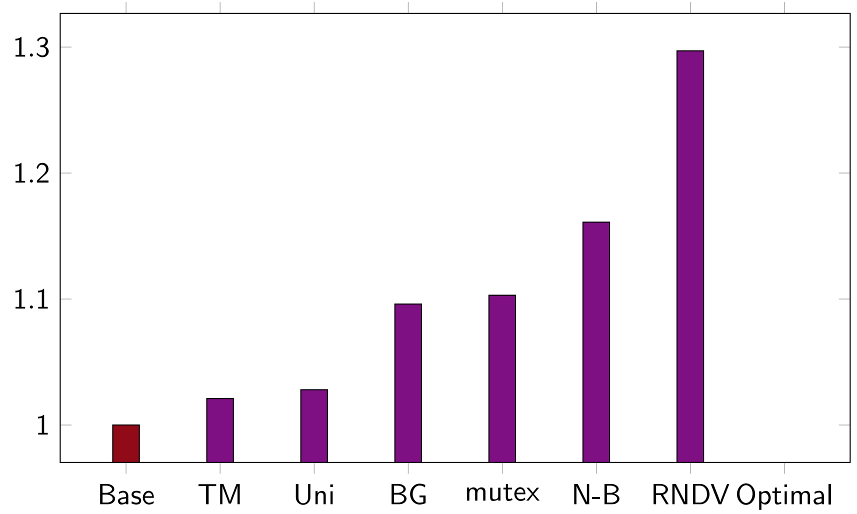


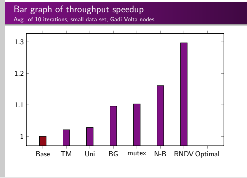
nevikw39

This is the baseline.

Bar graph of throughput speedup

Avg. of 10 iterations, small data set, Gadi Volta nodes



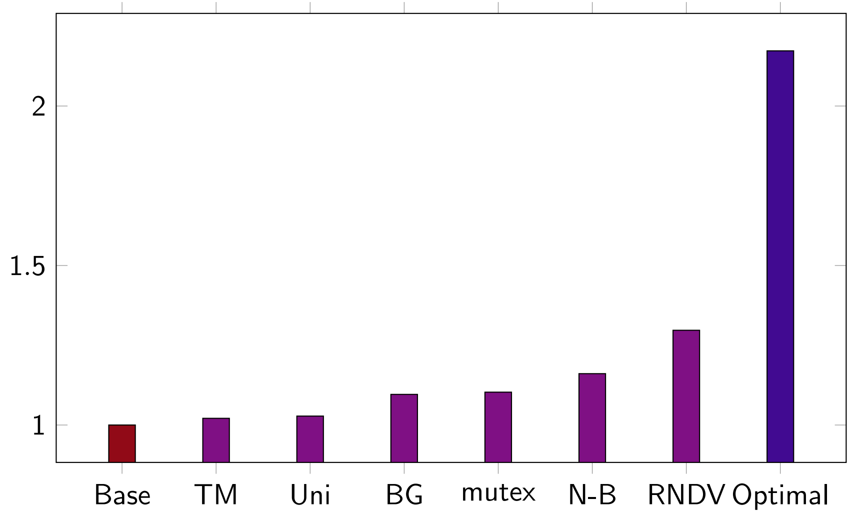


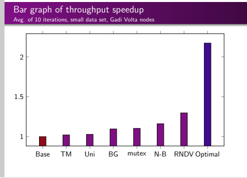
nevikw39

These are the speedup of a single option resp.

Bar graph of throughput speedup

Avg. of 10 iterations, small data set, Gadi Volta nodes





`nevikw39`

As we can easily find, the optimized combination of configs give rise to as significant improvement of throughput.

Overall Throughput Result on DGX-A100 nodes

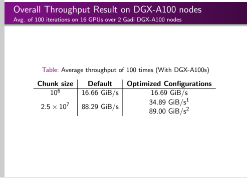
Avg. of 100 iterations on 16 GPUs over 2 Gadi DGX-A100 nodes

Table: Average throughput of 100 times (With DGX-A100s)

Chunk size	Default	Optimized Configurations
10^6	16.66 GiB/s	16.69 GiB/s
2.5×10^7	88.29 GiB/s	34.89 GiB/s[1] 89.00 GiB/s[2]

2022-10-24

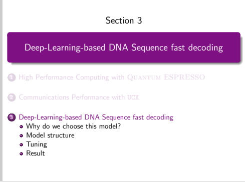
2022 APAC HPC-AI
└─Communications Performance with UCX
  └─Running on DGX-A100
    └─Overall Throughput Result on DGX-A100 nodes


Overall Throughput Result on DGX-A100 nodes
Avg. of 100 iterations on 16 GPUs over 2 Gadi DGX-A100 nodes
Table: Average throughput of 100 times (With DGX-A100s)
Chunk size | Default | Optimized Configurations
10^6 | 16.66 GiB/s | 16.69 GiB/s
2.5×10^7 | 88.29 GiB/s | 34.89 GiB/s[1] 89.00 GiB/s[2]

Section 3

2022-10-24

2022 APAC HPC-AI
└─Deep-Learning-based DNA Sequence fast decoding


Section 3

Deep-Learning-based DNA Sequence fast decoding

1 High Performance Computing with QUANTUM ESPRESSO

2 Communications Performance with UCX

3 Deep-Learning-based DNA Sequence fast decoding
- Why do we choose this model?
- Model structure
- Tuning
- Result

More accurate, more efficiency

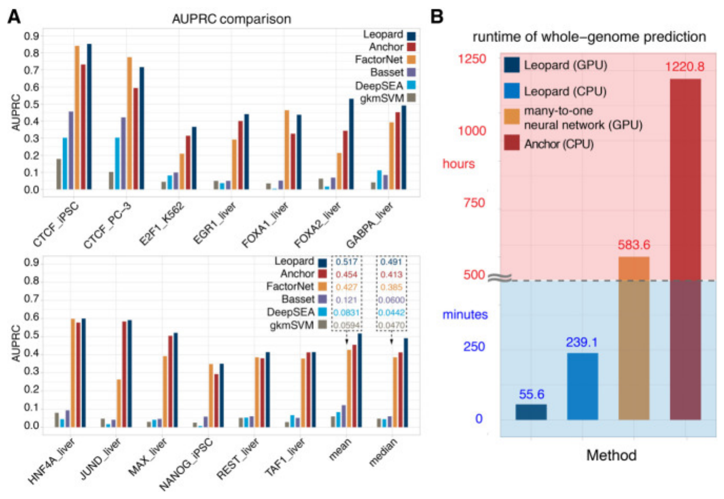
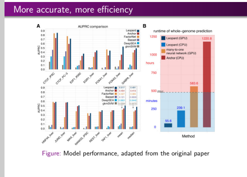


Figure: Model performance, adapted from the original paper

2022-10-24

2022 APAC HPC-AI
Deep-Learning-based DNA Sequence fast decoding
Why do we choose this model?
More accurate, more efficiency

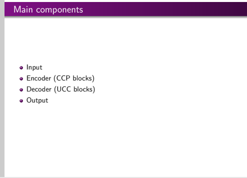


Main components

- Input
- Encoder (CCP blocks)
- Decoder (UCC blocks)
- Output

2022-10-24

2022 APAC HPC-AI
└Deep-Learning-based DNA Sequence fast decoding
└Model structure
└Main components


Main components

- Input
- Encoder (CCP blocks)
- Decoder (UCC blocks)
- Output

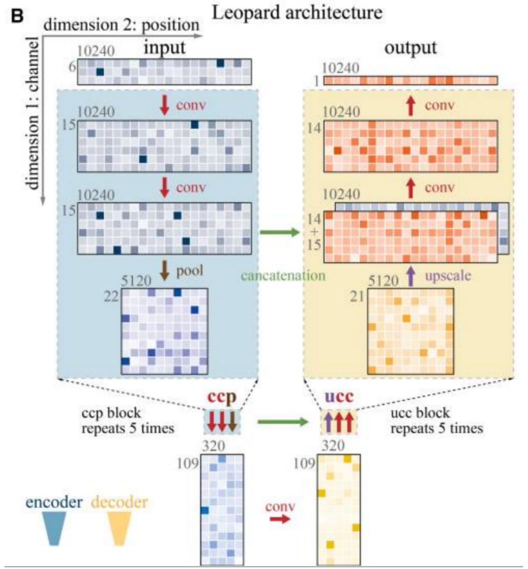
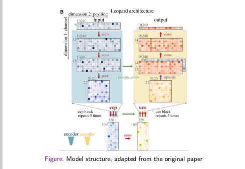


Figure: Model structure, adapted from the original paper

2022-10-24

2022 APAC HPC-AI
└Deep-Learning-based DNA Sequence fast decoding
 └Model structure

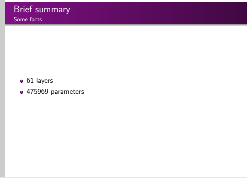


Brief summary
Some facts

- 61 layers
- 475969 parameters

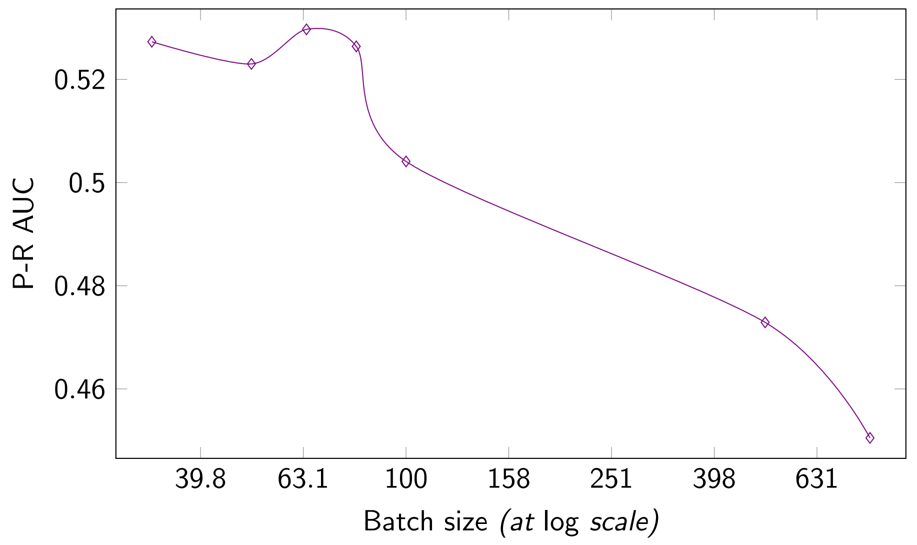
2022-10-24

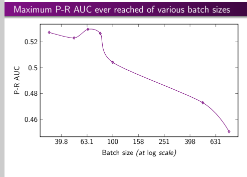
2022 APAC HPC-AI
└─Deep-Learning-based DNA Sequence fast decoding
  └─Model structure
    └─Brief summary


Brief summary
Some facts

- 61 layers
- 475969 parameters

Maximum P-R AUC ever reached of various batch sizes





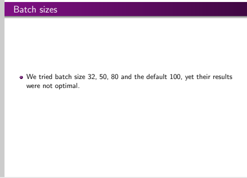
In our experiments, we follow a hierarchical manner; that is, we first tested **batch size**, then we adjusted **learning rate** for a particular **batch size**.

Batch sizes

- We tried batch size 32, 50, 80 and the default 100, yet their results were not optimal.

2022-10-24

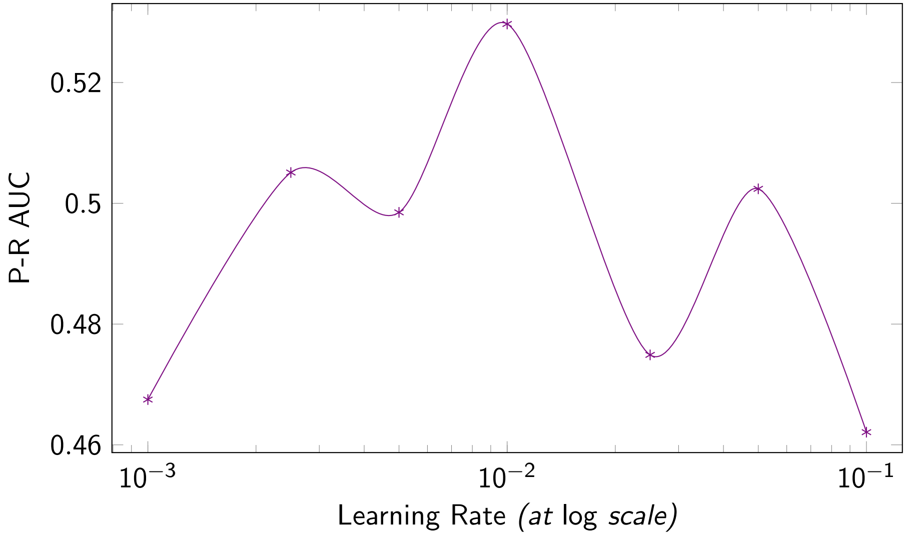
2022 APAC HPC-AI
└Deep-Learning-based DNA Sequence fast decoding
  └Tuning
    └Batch sizes

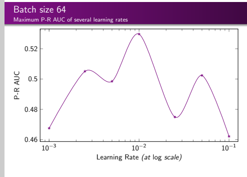

Batch sizes

- We tried batch size 32, 50, 80 and the default 100, yet their results were not optimal.

Batch size 64

Maximum P-R AUC of several learning rates

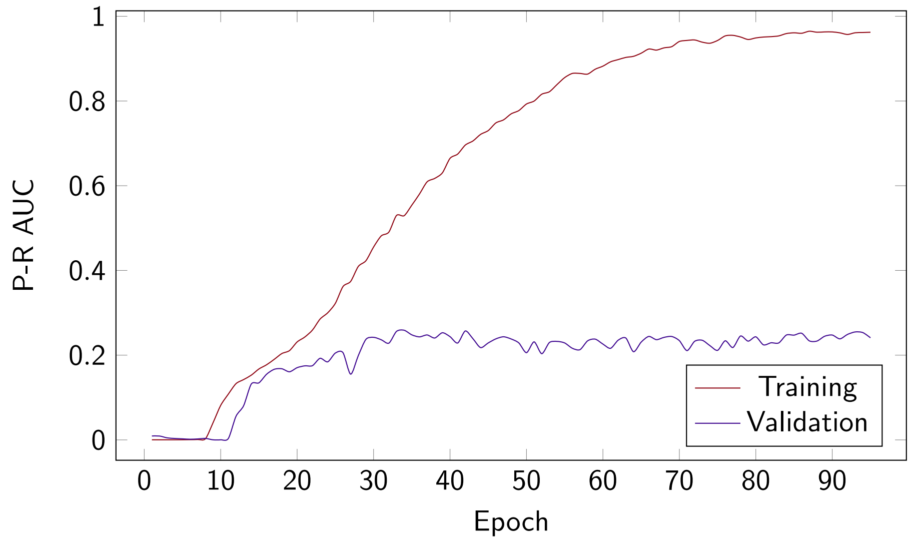


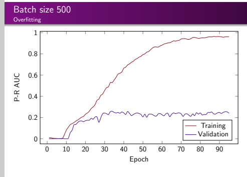


This is the best **batch size** we have ever found. We tested several **learning rate**, such as 0.001, 0.0025, 0.005, 0.01, 0.025, 0.05, among of which 0.01 gave rise to optimal result.

Batch size 500

Overfitting

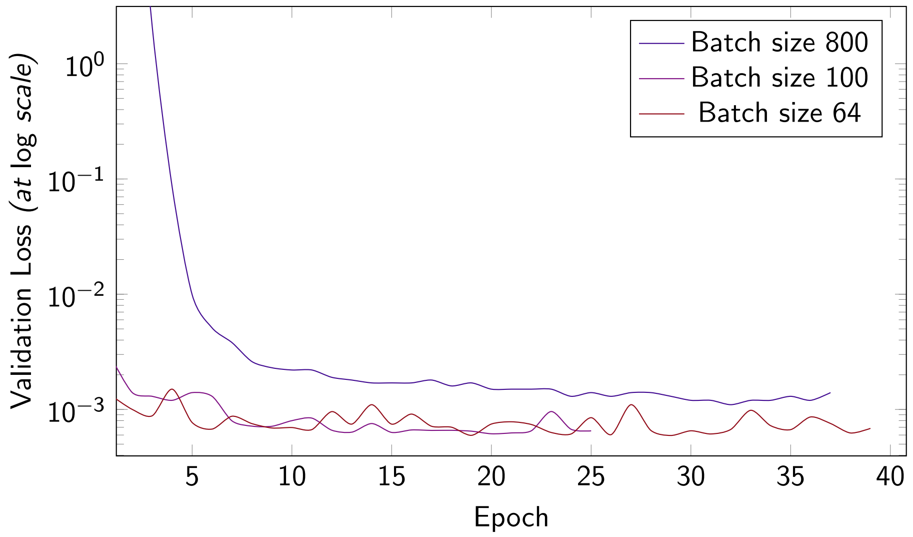




We encounter severe **overfitting** when it comes to this **batch size**; that is, *P-R AUC* could increase up to about 0.96 for the training set, whereas the ones for validation set were as low as 0.25. What's worse, since we increased the patience of early stop to 10, it exhausted the 2-hour wall time.

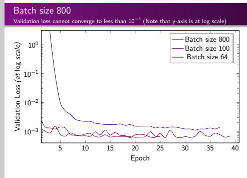
Batch size 800

Validation loss cannot converge to less than 10^{-3} (Note that y-axis is at log scale)



2022-10-24

2022 APAC HPC-AI
└Deep-Learning-based DNA Sequence fast decoding
└Tuning
└Batch size 800



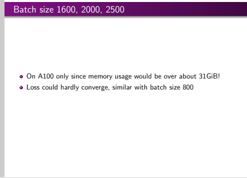
The maximum **batch size** for V100 is 800 since it consumes about 31GB GPU memory. For this **batch size**, the converge rate of validation loss were quite slow despite of **learning rate**. As a consequence, the results were also not so good.

Batch size 1600, 2000, 2500

- On A100 only since memory usage would be over about 31GiB!
- Loss could hardly converge, similar with batch size 800

2022-10-24

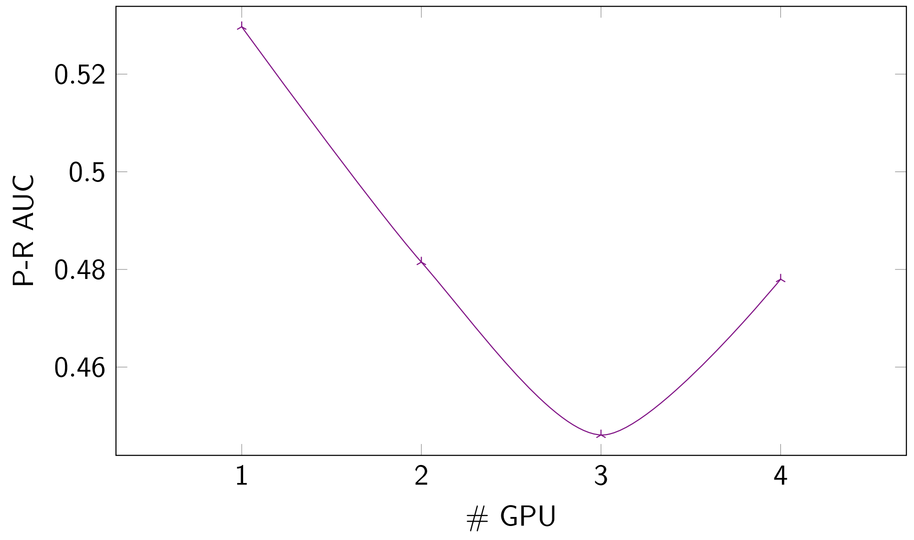
2022 APAC HPC-AI
└─Deep-Learning-based DNA Sequence fast decoding
  └─Tuning
    └─Batch size 1600, 2000, 2500

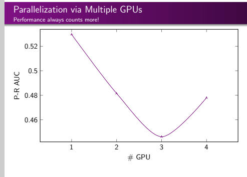

Batch size 1600, 2000, 2500

- On A100 only since memory usage would be over about 31GiB!
- Loss could hardly converge, similar with batch size 800

Parallelization via Multiple GPUs

Performance always counts more!



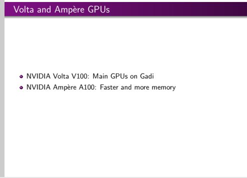


We also managed to parallelize the training and utilized multiple GPUs. Nonetheless, the performance would decline a lot. Furthermore, the metrics would drop suddenly and dramatically after the number of GPU was over a value. We guessed that it might due to the simple, linear scale of **learning rate**, trying to find ad-hoc optimal **learning rate** for different number of GPU yet in vain.

Eventually, we decide to train on only a single GPU since we believe that the performance counts more. Still, we provide the job script to train across two A100s.

Volta and Ampère GPUs

- NVIDIA Volta V100: Main GPUs on Gadi
- NVIDIA Ampère A100: Faster and more memory


Volta and Ampère GPUs

- NVIDIA Volta V100: Main GPUs on Gadi
- NVIDIA Ampère A100: Faster and more memory

Hyperparameters and environments

Batch Size	64
Learning Rate	0.01
# GPU	1

2022-10-24

2022 APAC HPC-AI
└Deep-Learning-based DNA Sequence fast decoding
  └Result
    └Hyperparameters and environments

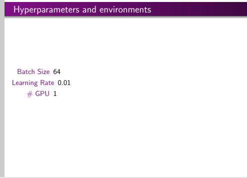


Table: Performance of Single GPU

GPU Type	Loss *(Binary Crossentropy)*	P-R AUC	Dice coeff.	Binary Int. of Union	Training Time [s]
V100	7.2768×10^{-4}	0.5297	0.3705	0.3625	3170.19
A100	7.3153×10^{-4}	0.5288	0.4109	0.3701	941.63

2022-10-24

2022 APAC HPC-AI
└─Deep-Learning-based DNA Sequence fast decoding
└─Result

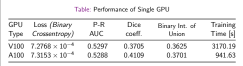


Table: Performance of Single GPU

GPU Type	Loss *(Binary Crossentropy)*	P-R AUC	Dice coeff.	Binary Int. of Union	Training Time [s]
V100	7.2768×10^{-4}	0.5297	0.3705	0.3625	3170.19
A100	7.3153×10^{-4}	0.5288	0.4109	0.3701	941.63