MS52
Next Generation FFT Algorithms in Theory and Practice: Parallel Implementations and Applications

- **Organizers:**
  – **Daisuke Takahashi**
    University of Tsukuba, Japan
  – **Franz Franchetti**
    Carnegie Mellon University, U.S.
  – **Samar A. Aseeri**
    King Abdullah University of Science & Technology (KAUST), Saudi Arabia

Aim of this minisymposium

- The fast Fourier Transform (FFT) is an algorithm used in a wide variety of applications, yet does not make optimal use of many current hardware platforms.
- Hardware utilization performance, on its own, does not however, imply optimal problem solving.
- The purpose of this mini-symposium is to enable the exchange of information between people working on alternative FFT algorithms, to those working on FFT implementations, in particular for parallel hardware.
- In addition to FFT algorithms, number-theoretical transform (NTT) is also included in the topic of this minisymposium.
- http://www.fft.report

MS52

- **3:45-4:10 Implementation of Parallel Number-Theoretic Transform on GPU Clusters**

  Daisuke Takahashi, University of Tsukuba, Japan
- **4:15-4:40 FFTX: Release, Updates and Next Steps**

  Franz Franchetti and *Sanil Rao,* Carnegie Mellon University, U.S.
- **4:45-5:10 A Comparison of Intel and OSU All-to-all Benchmarks for Next Generation FFT Algorithms**

  Samar A. Aseeri, King Abdullah University of Science & Technology (KAUST), Saudi Arabia; Benson Muite, Kichakato Kizito, Kenya; David E. Keyes, KAUST, Saudi Arabia and Columbia University, U.S.
- **5:15-5:40 Latest Advanced on Parallel and Distributed FFT Computation on NVIDIA GPU**

  Miguel Ferrer Avila, Josh Romero, Lukasz Ligowski, and Filippo Spiga, NVIDIA, U.S.

Implementation of Parallel Number-Theoretic Transform on GPU Clusters

Daisuke Takahashi
Center for Computational Sciences
University of Tsukuba, Japan

Outline

- Background
- Objectives
- Number-theoretic Transform (NTT)
- Four-Step NTT Algorithm
- Parallel Implementation of NTT
- Performance Results
- Conclusion

Background

- The fast Fourier transform (FFT) is an algorithm that is widely used today in scientific and engineering computing.
- FFTs are often computed using complex or real numbers, but it is known that these transforms can also be computed in a ring and a finite field [Pollard 1971].
- Such a transform is called the number-theoretic transform (NTT).
- The NTT is used for homomorphic encryption, polynomial multiplication, and multiple-precision multiplication.

Related Works (1/2)

- Spiral-generated modular FFTs have been proposed [Meng et al. 2010 and 2013].
  - Experiments were performed using 32-bit integers and 16-bit primes with Intel SSE4 instructions.
- An implementation of NTT using the Intel AVX-512IFMA (Integer Fused Multiply-Add) instructions has been proposed [Boemer et al. 2021].
  - This implementation is available as the Intel Homomorphic Encryption (HE) Acceleration Library.
  - Intel HEXL targets the typical data size $n = [2^{10}, 2^{17}]$ of NTTs used in homomorphic encryption and is not parallelized.

Related Works (2/2)

- An Implementation of Parallel Number-Theoretic Transform Using Intel AVX-512 Instructions has been proposed [Takahashi 2022].
  – NTT kernels are vectorized using the Intel AVX-512 instructions.
  – Six-step NTT is parallelized using OpenMP.
- Vectorizing and distributing number-theoretic transform on Arm-based supercomputers have been proposed [Jesus et al. 2023].
  – For counting Goldbach partitions.

Objectives

- We consider accelerating NTT for larger data sizes by parallelization, targeting polynomial multiplication and multiple-precision multiplication.
- We parallelize the four-step NTT using MPI and OpenACC.

Number-Theoretic Transform (NTT)

- The number-theoretic transform (NTT) can be expressed in a field $\mathbf{F}_p = \mathbf{Z}/p\mathbf{Z}$, where p is a prime number:

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk} \bmod p, \quad 0 \le k \le n-1,$$

in which ω_n is the primitive n-th root of unity.

- The n-point NTT is directly computed by $O(n^2)$ arithmetic operations, but by applying an algorithm similar to FFT, the number of arithmetic operations can be reduced to $O(n \log n)$.

Stockham Radix-2 NTT Algorithm

Algorithm 1 Stockham radix-2 NTT algorithm

Input: $n = 2^q,\ X_0(j) = x(j),\ 0 \le j \le n-1$, and
ω_n is the primitive n-th root of unity.

Output: $y(k) = X_q(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk} \bmod p,\ 0 \le k \le n-1$

$l \leftarrow n/2$
$m \leftarrow 1$
for t **from** 1 **to** q **do**
  for j **from** 0 **to** $l-1$ **do**
    for k **from** 0 **to** $m-1$ **do**
      $c_0 \leftarrow X_{t-1}(k + jm)$
      $c_1 \leftarrow X_{t-1}(k + jm + lm)$
      $X_t(k + 2jm) \leftarrow (c_0 + c_1) \bmod p$
      $X_t(k + 2jm + m) \leftarrow \omega_n^{jm}(c_0 - c_1) \bmod p$
    end for
  end for
  $l \leftarrow l/2$
  $m \leftarrow 2m$
end for

Modular Arithmetic in NTT

- The butterfly operation of the NTT can be performed using modular addition, subtraction, and multiplication.
- The modular addition $c = (a + b) \bmod N$ for $0 \leq a, b < N$ can be replaced by the addition $c = a + b$ and the conditional subtraction $c - N$ when $c \geq N$.
- Modular multiplication includes modulo operations, which are slow due to the integer division process.
- However, Montgomery multiplication [Montgomery 1985] and Shoup's modular multiplication [Harvey 2014] are known to avoid this problem.

Montgomery Multiplication Algorithm [Montgomery 1985]

Algorithm 2 Montgomery multiplication algorithm

Input: A, B, N such that $0 \le A, B < N$, $\beta > N$, $\gcd(\beta, N) = 1$, $\mu = -N^{-1} \bmod \beta$

Output: $C = AB\beta^{-1} \bmod N$ such that $0 \le C < N$

1: $C \leftarrow AB$
2: $q \leftarrow \mu C \bmod \beta$
3: $C \leftarrow (C + qN)/\beta$
4: **if** $C \ge N$ **then**
5: $\quad C \leftarrow C - N$
6: **return** C.

Modular Multiplication

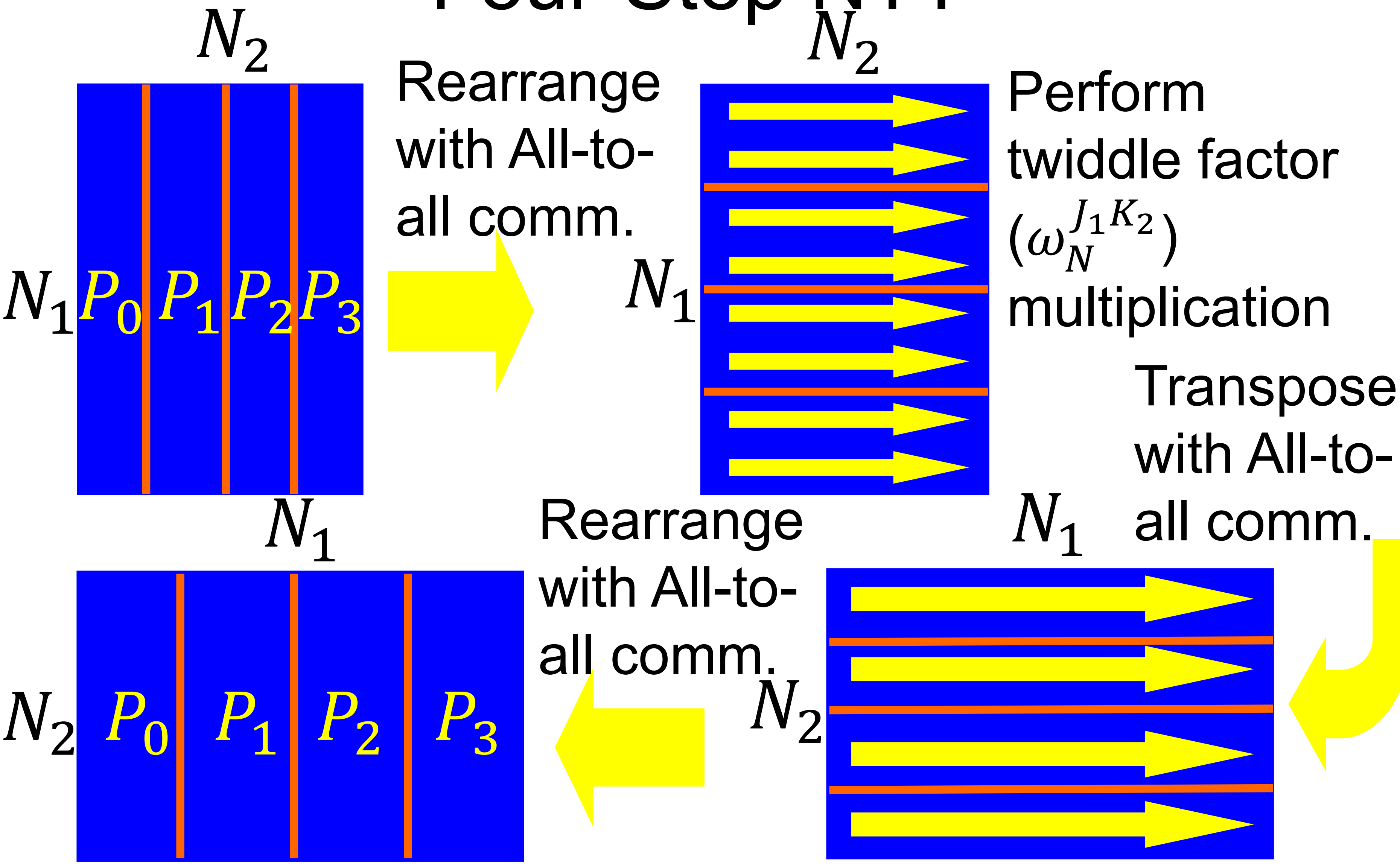
- The modular multiplication $c = ab \bmod N$ can be performed using Montgomery multiplication as follows:
- Convert a and b to Montgomery representations $A = a\beta \bmod N$ and $B = b\beta \bmod N$, where β is an integer such that $\beta > N$ and $\gcd(\beta, N) = 1$.
- Perform Montgomery multiplication $C = AB\beta^{-1} \bmod N$, where β^{-1} is the modular multiplicative inverse of $\beta\beta^{-1} \equiv 1 \pmod{N}$.
- Inverse transforming the results of Montgomery multiplication C to its representation c in the original domain:

$$
\begin{aligned}
c &= C\beta^{-1} \bmod N = (AB\beta^{-1} \bmod N)\,\beta^{-1} \bmod N \\
&= \{(a\beta \bmod N)(b\beta \bmod N)\beta^{-1} \bmod N\}\beta^{-1} \bmod N \\
&= ab \bmod N
\end{aligned}
$$

Four-Step NTT Algorithm

- If n has factors n_1 and n_2 $(n = n_1 \times n_2)$, in the same way as the four-step FFT algorithm [Bailey 1990], the following four-step NTT algorithm is derived:
- Step 1: n_1 simultaneous n_2-point multirow NTTs
- Step 2: Twiddle factor ($\omega_n^{j_1 k_2}$) multiplication
- Step 3: Transposition
- Step 4: n_2 simultaneous n_1-point multirow NTTs

Parallel NTT Algorithm Based on Four-Step NTT



Parallelization of Four-Step NTT

```
#pragma acc data present(a[0:nn], b[0:nn], wx[0:nx/2], wy[0:ny/2], w[0:nn]) {
/* Step 1: Rearrange (nx / nproc) * nproc * (ny / nproc)
            to (nx / nproc) * (ny / nproc) * nproc */
#pragma acc parallel loop collapse(3)
 for (k = 0; k < nproc; k++)
   for (j = 0; j < nny; j++)
     for (i = 0; i < nnx; i++)
       b[i + j * nnx + k * (nnx * nny)] = a[i + k * nnx + j * (nnx * nproc)];
/* Step 2: All-to-all communication */
#pragma acc host_data use_device(a, b)
 MPI_Alltoall(b, nn / nproc, MPI_UNSIGNED_LONG_LONG, a, nn / nproc,
              MPI_UNSIGNED_LONG_LONG, MPI_COMM_WORLD);
/* Step 3: (nx / nproc) simultaneous ny-point multirow NTTs */
 nttsub(a, b, wy, nnx, ny, ipy, np, mu);
…
```

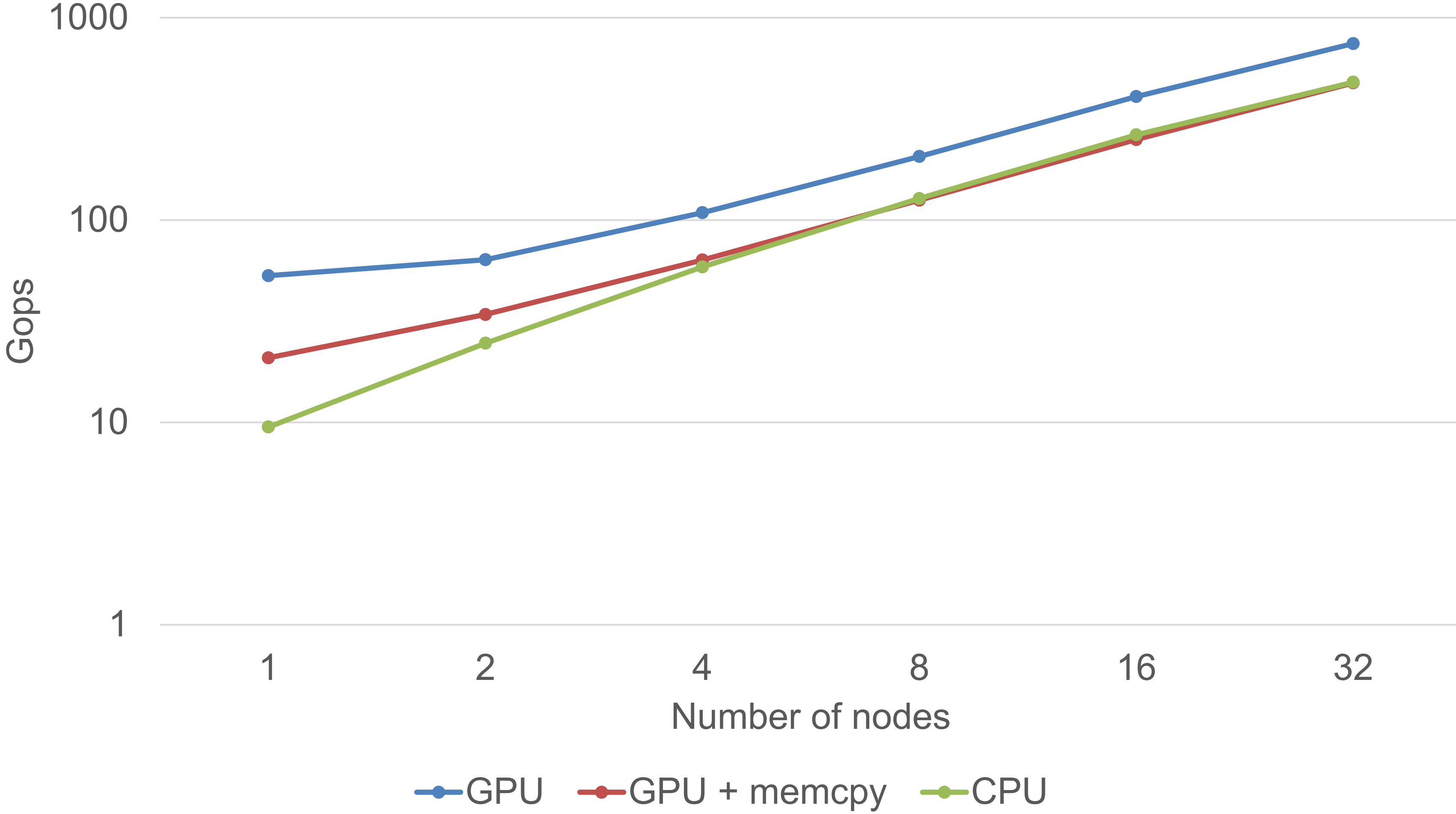
Performance Results

- For performance evaluation, we compared the performance of the following parallel NTTs with a modulus of 63 bits:
  - MPI+OpenACC (GPU implementation) of the four-step NTT
  - MPI+OpenMP (CPU implementation) of the six-step NTT [Takahashi 2022]
- The giga-operations per second (Gops) values are each based on $(3/2)N\log_2 N$ for a transform of size $N = 2^m$.

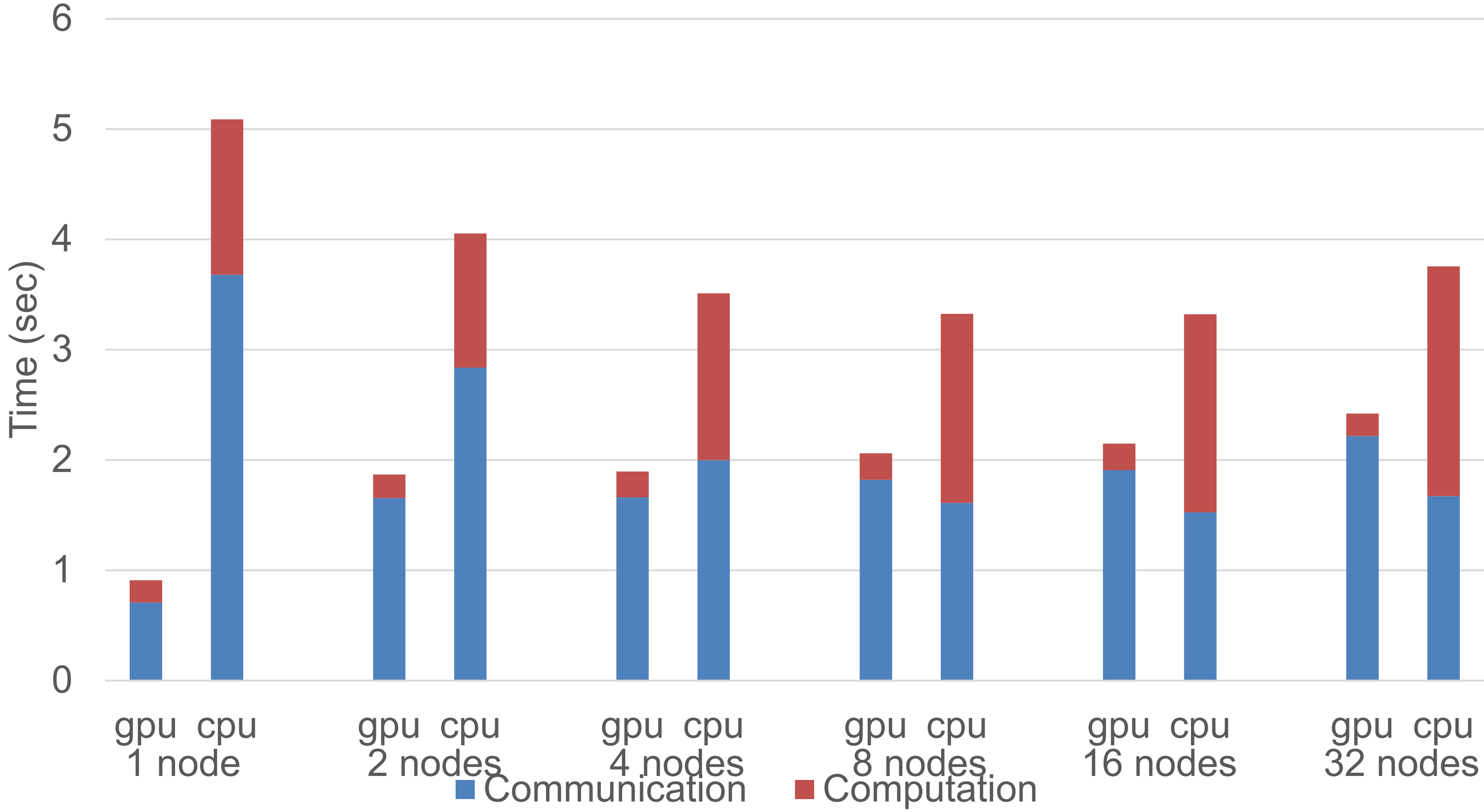
Evaluation Environment

- The performance was measured on the Pegasus, a GPU cluster at the University of Tsukuba.
  - 120 nodes, Peak 6.5 PFlops
  - CPU: Intel Xeon Platinum 8468
    (48 cores, 2.1 GHz, 3.2 TFlops)
  - GPU: NVIDIA H100 Tensor Core GPU with PCIe
  - Interconnect: NVIDIA Quantum-2 InfiniBand (200 Gbps)
  - Compiler: NVIDIA HPC Compilers 23.9
  - MPI library: OpenMPI 4.1.5
  - Compiler option:
    “-fast -acc=gpu -gpu=cc90 (for GPU implementation)
    “-fast -mp -tp=sapphirerapids (for CPU implementation)
- Each node has 48 cores and 1 MPI process.

Performance of Parallel NTTs ($N = 2^{30} \times$ number of nodes)



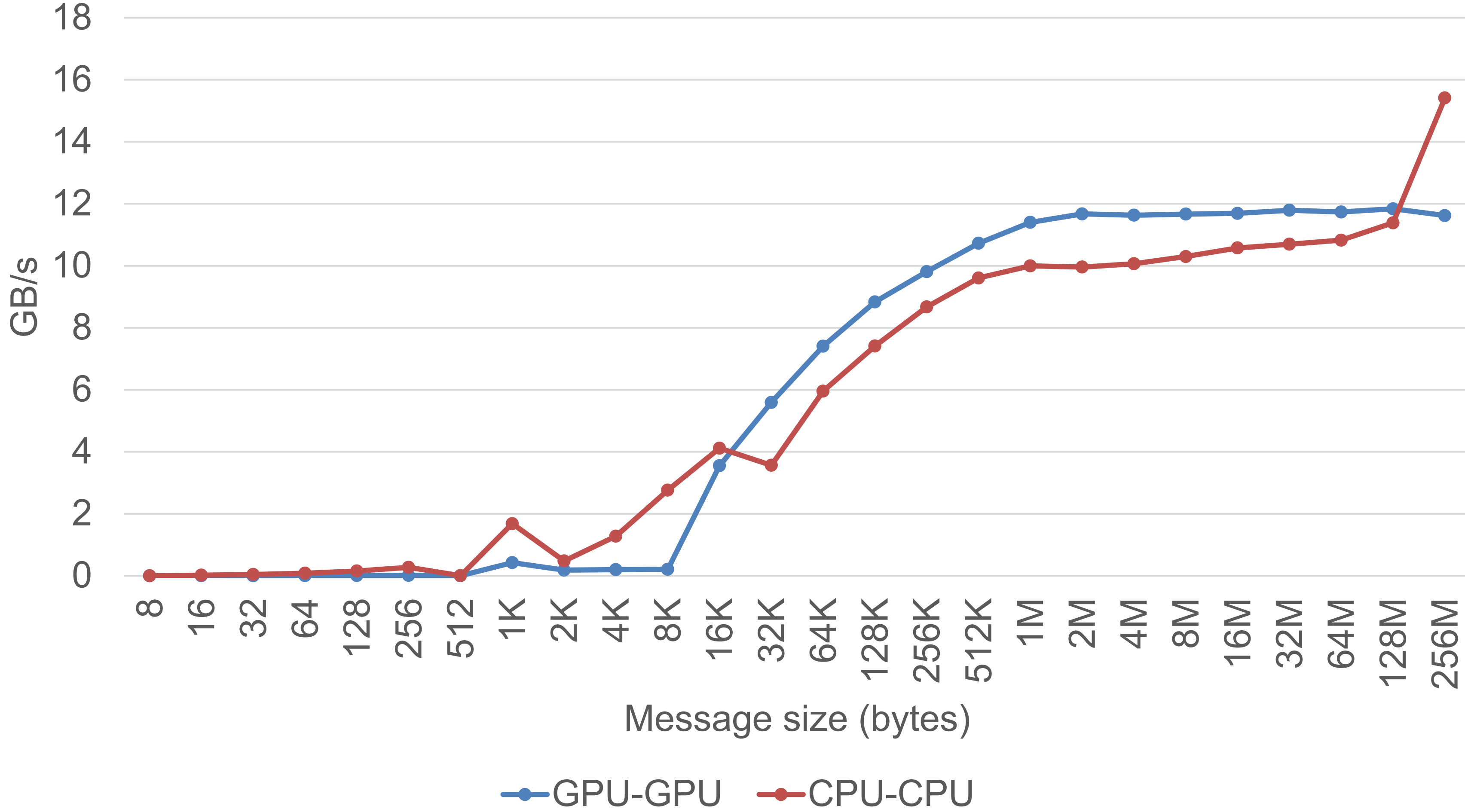
Breakdown of Execution Time of GPU and CPU implementations ($N = 2^{30} \times$ number of nodes)



Discussion

- In the case of using GPUs, the computation time is reduced as compared with the case of using CPUs only, whereas the communication time is almost the same.
- We can clearly see that the all-to-all communication overhead contributes significantly to the execution time.
- For this reason, the difference in performance between GPU implementation and CPU implementation decreases as the number of nodes increases.
- PCIe transfer is the chief bottleneck because the bandwidth of PCIe Gen 5 is only 128 GB/s, whereas the memory bandwidth of NVIDIA H100 Tensor Core GPU with PCIe is 2000 GB/s.

Performance of All-to-all Communication (32 nodes, 32 MPI processes)



Conclusion

- We proposed the implementation of the parallel NTT on GPU clusters.
- The butterfly operation of the NTT can be performed using modular addition, subtraction, and multiplication.
- We parallelized the four-step NTT using MPI and OpenACC.
- We successfully achieved a performance of over 745 Gops on 32 nodes of the Pegasus (120 nodes) for a 2^{35}-point NTT with a modulus of 63 bits.