Sandia National Laboratories

Exceptional service in the national interest

TEMPI: An Interposed MPI Library with a Canonical Representation of CUDA-aware Datatypes

Carl Pearson[1], Kun Wu[2], I-Hsin Chung[3], Jinjun Xiong[3], Wen-Mei Hwu[4]

[1]Sandia National Labs / [2]University of Illinois Electrical and Computer Engineering / [3]IBM T. J. Watson Research / [4]Nvidia Research

HPDC - Jun 24, 2021

Work completed at University of Illinois prior to joining Sandia National Labs

SAND2021-6846 C

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

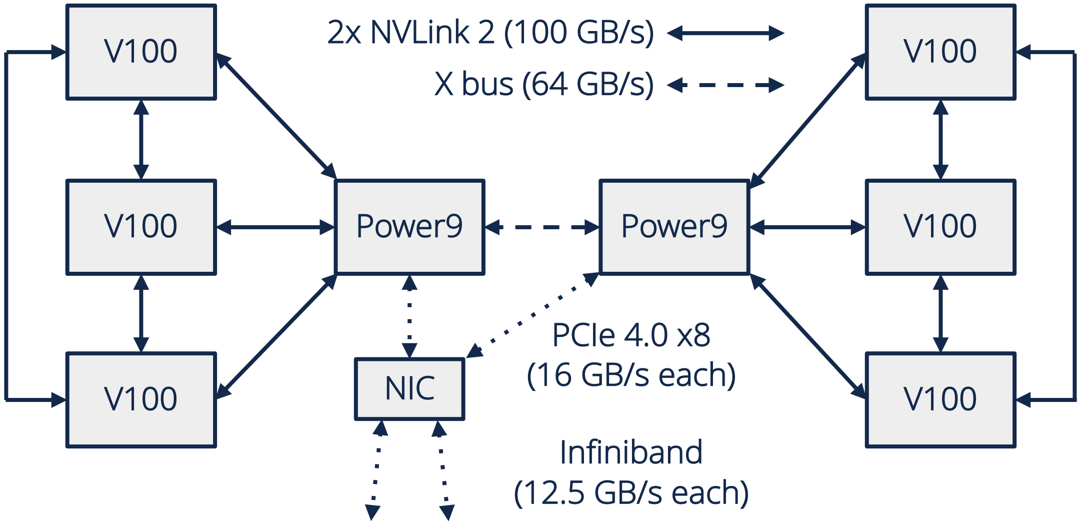
U.S. DEPARTMENT OF ENERGY

NNSA

Outline

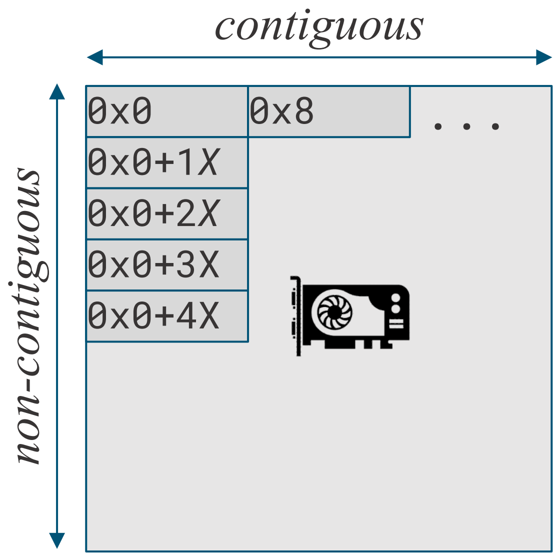
- Distributed GPU stencil, non-contiguous data
- Equivalence of strided datatypes and minimal representation
- GPU communication methods
- Deploying on managed systems
- Large messages and MPI datatypes
- Translation and canonicalization
- Automatic model-driven transfer method selection
- Interposed library implementation

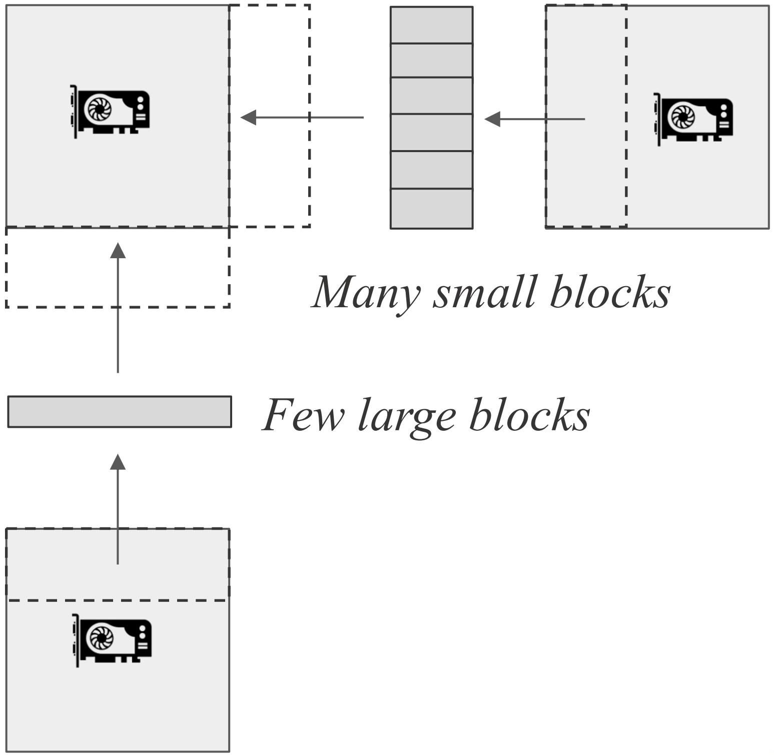
OLCF Summit Node



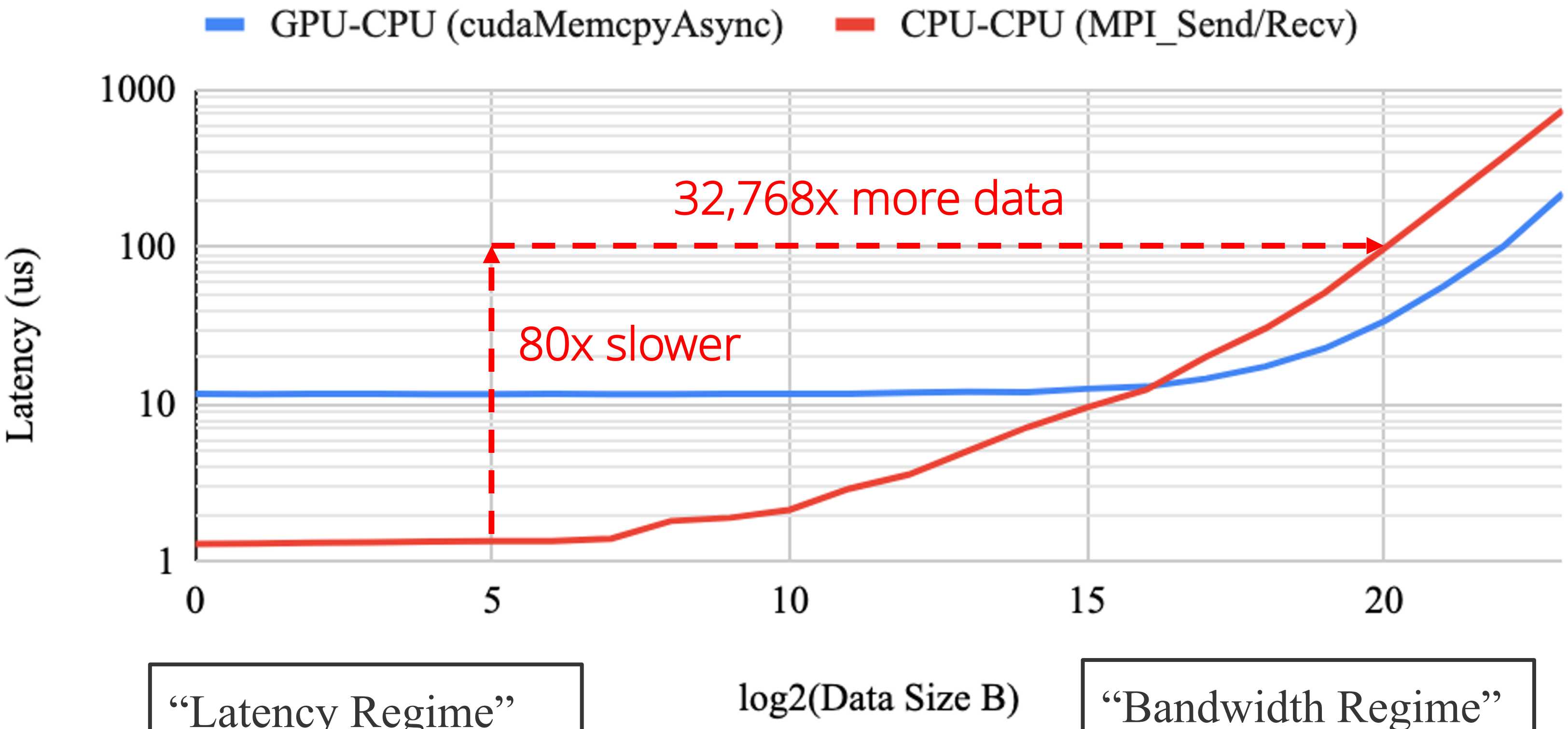
Summit Node
(bidirectional bandwidth)

Stencil Communication and non-contiguous Data





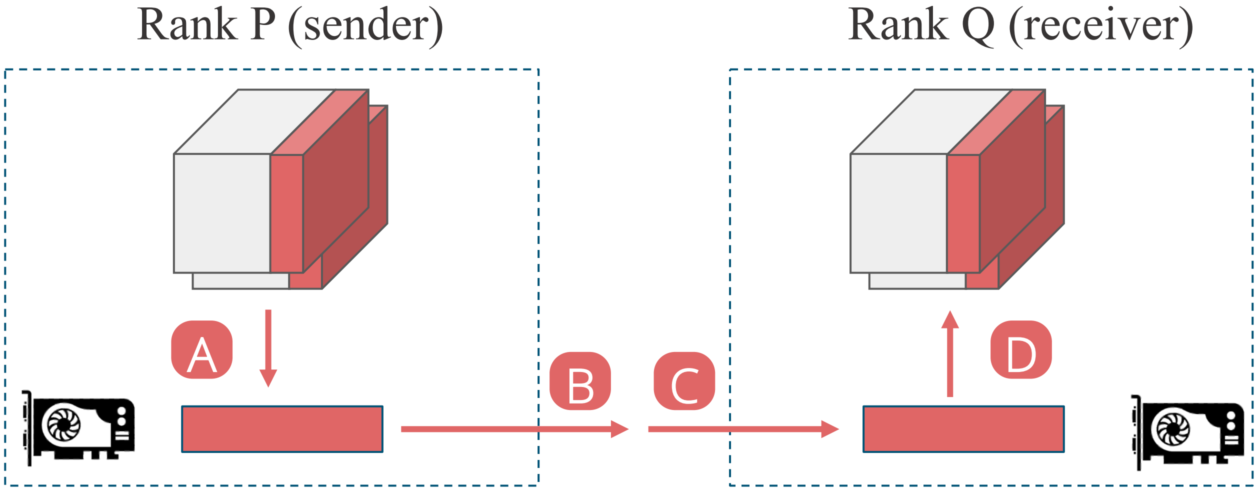
OLCF Summit Latency vs Transfer Size



"Latency Regime"

"Bandwidth Regime"

CUDA-Aware MPI + Packing



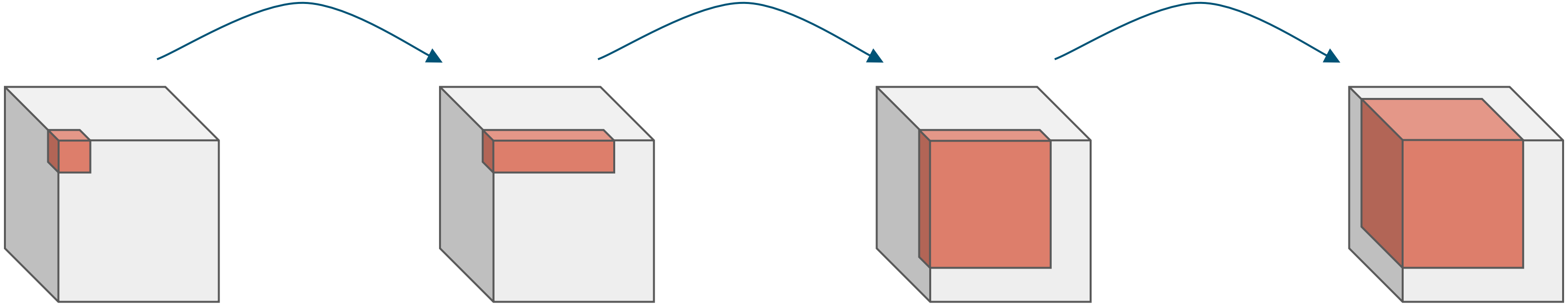
A pack<<<>>>

B MPI_Isend

C MPI_Irecv

D unpack<<<>>>

MPI Derived Datatypes



"MPI_Type_contiguous(…)"

"MPI_Type_vector(…)"

"MPI_Type_vector(…)"

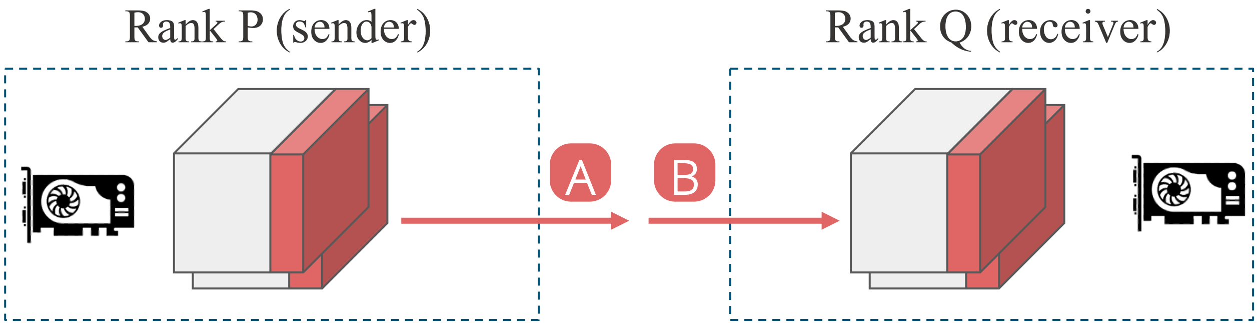
1 MPI_BYTE

row of contiguous bytes

plane of non-contiguous rows

cuboid of non-contigous planes

CUDA-Aware MPI + MPI Derived Datatypes



Setup (once):

Rank P (sender):

```
MPI_Type...
MPI_Type...
...
MPI_Type_commit
```

Rank Q (receiver):

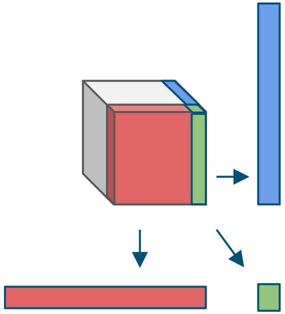
```
MPI_Type...
MPI_Type...
...
MPI_Type_commit
```

Each halo exchange:

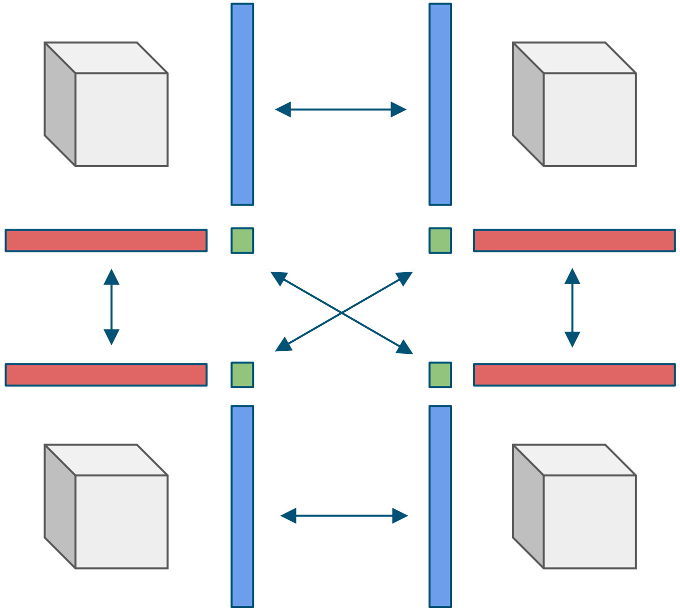
A `MPI_Isend`

B `MPI_Irecv`

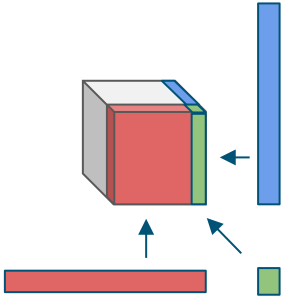
“alltoallv” Halo Exchange



MPI_Packs

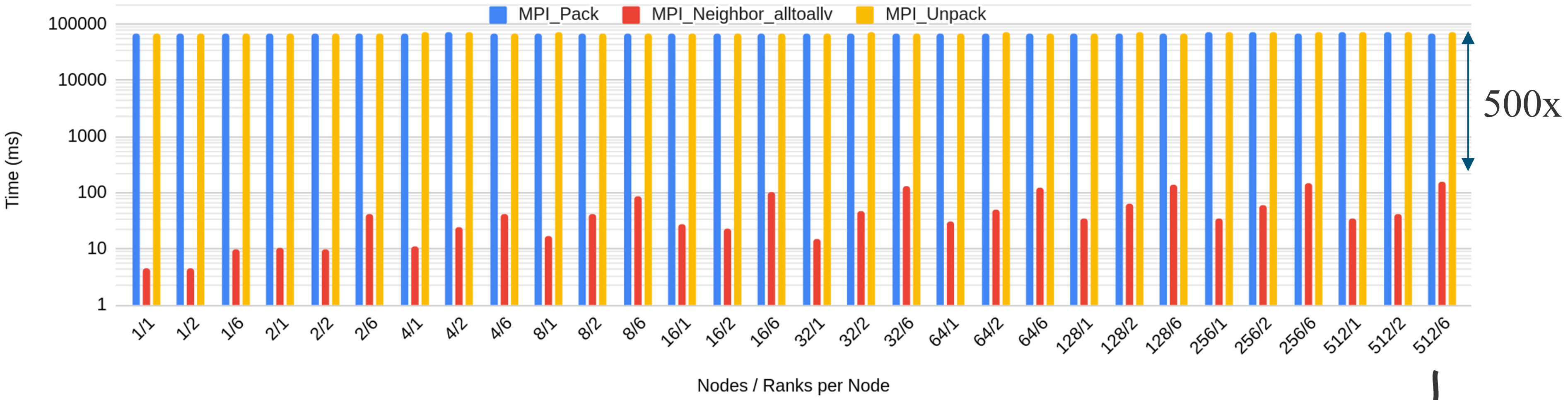


MPI_Neighbor_alltoallv



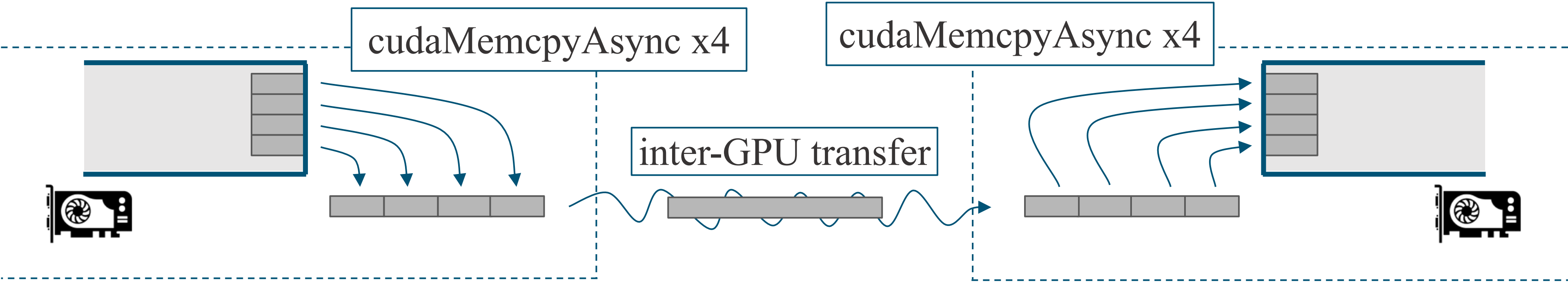
MPI_Unpacks

“alltoallv” with MPI derived types



- MPI_Neighbor_alltoallv = ~500 MB/s/rank
- MPI_Pack / MPI_Unpack = **~1 MB/s**

MPI_Send (Spectrum MPI)



cudaMemcpyAsync x4

inter-GPU transfer

cudaMemcpyAsync x4

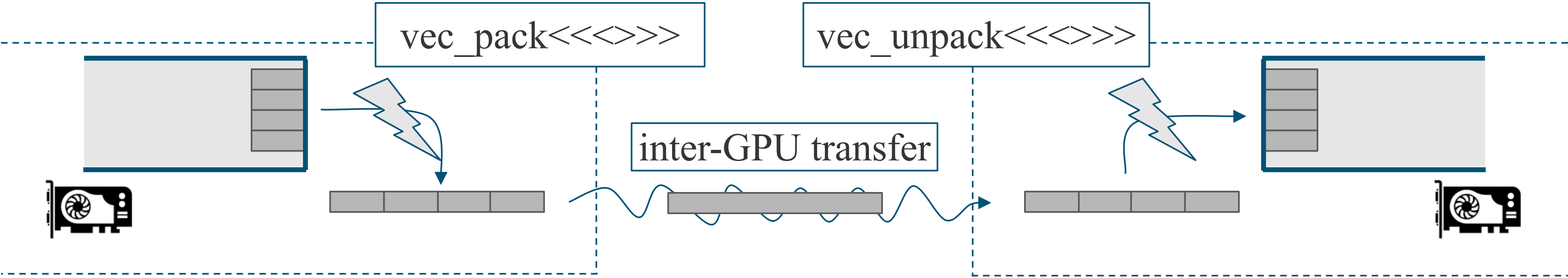
✗ “Latency Regime”

✅ “Bandwidth Regime”

✗ “Latency Regime”

Better MPI_Send (common foundation)

- GPU kernels to pack non-contiguous data



vec_pack<<<>>>

vec_unpack<<<>>>

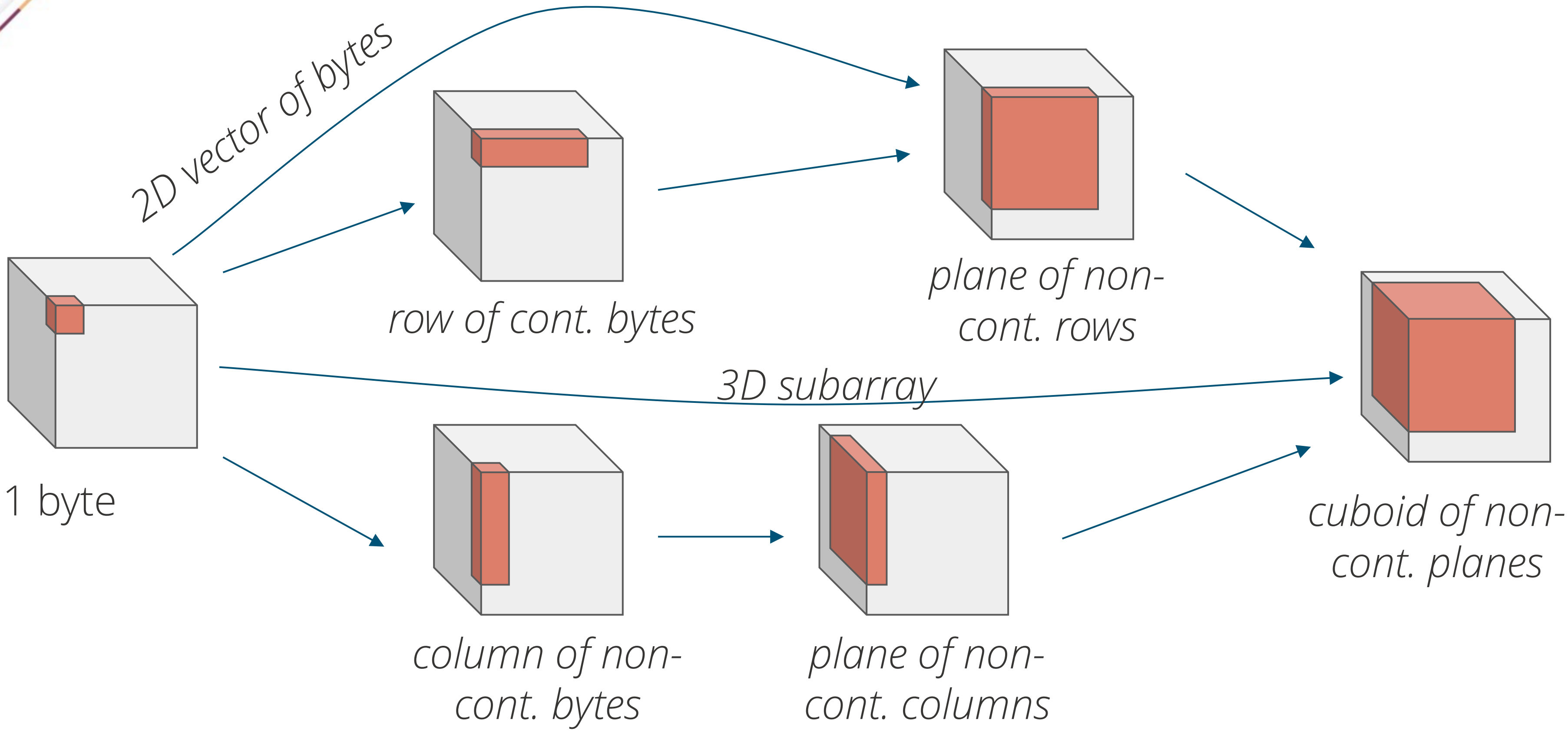
inter-GPU transfer

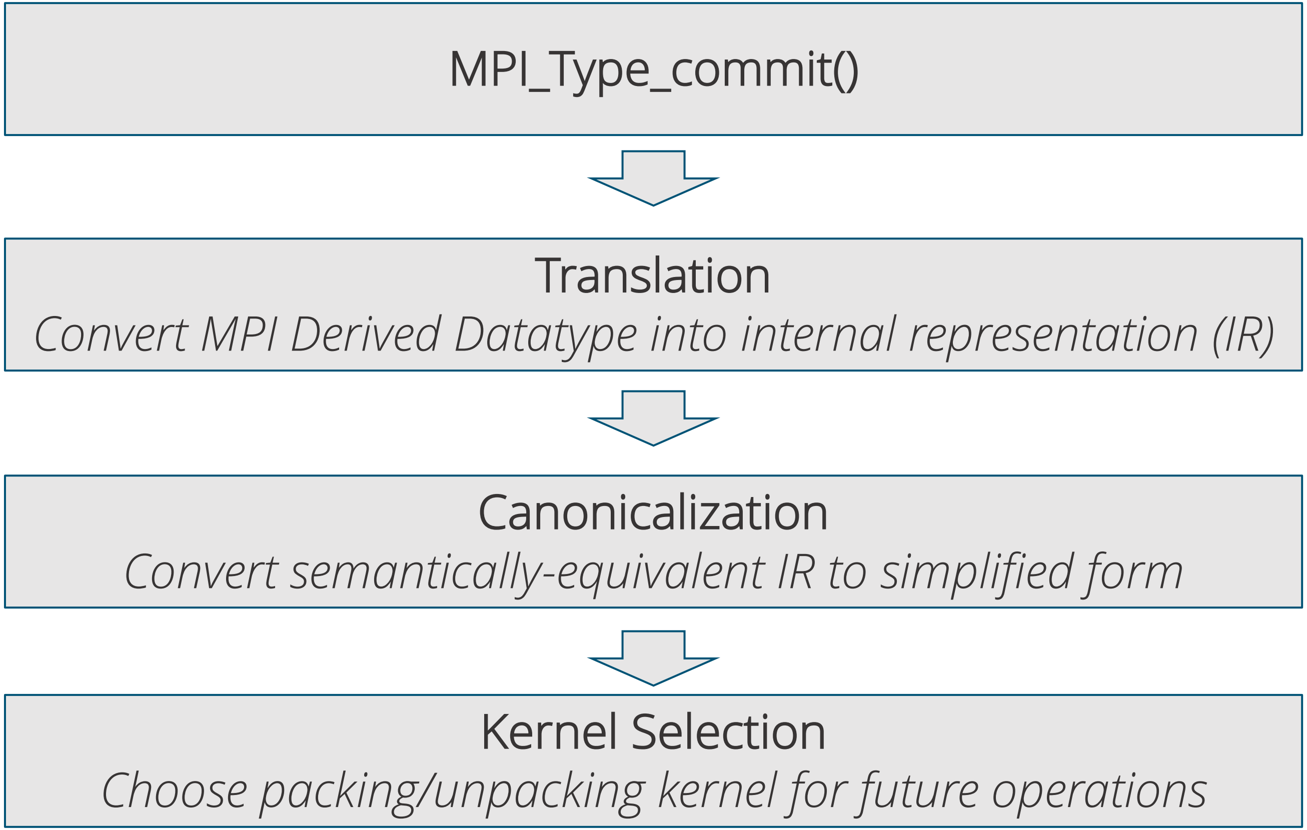
✅ “Bandwidth Regime”

✅ “Bandwidth Regime”

✅ “Bandwidth Regime”

MPI Derived Datatype Equivalence


2D vector of bytes
row of cont. bytes
plane of non-cont. rows
1 byte
3D subarray
cuboid of non-cont. planes
column of non-cont. bytes
plane of non-cont. columns



MPI_Type_commit()

Translation

Convert MPI Derived Datatype into internal representation (IR)

Canonicalization

Convert semantically-equivalent IR to simplified form

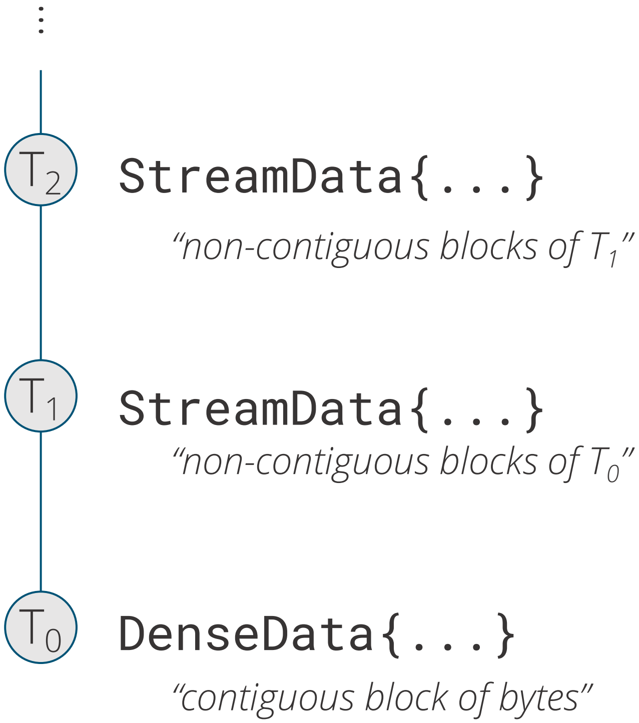
Kernel Selection

Choose packing/unpacking kernel for future operations

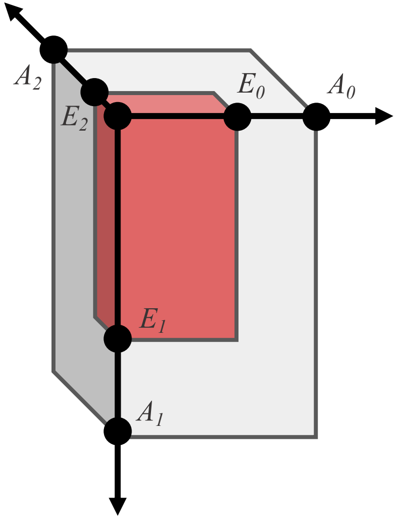
IR - “Internal Representation”

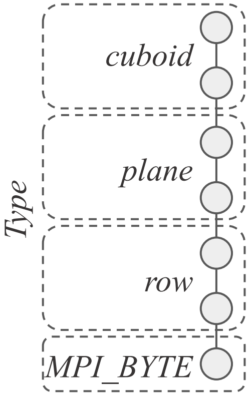
```
StreamData {
  integer offset; // offset (B) of the first element
  integer stride; // pitch (B) between element
  integer count;  // number of elements
}

DenseData {
  integer offset; // offset (B) of the first byte
  integer extent; // number of bytes
}
```

Hierarchy of StreamData, rooted at DenseData


⋮
T_2
StreamData{...}
“non-contiguous blocks of T_1”
T_1
StreamData{...}
“non-contiguous blocks of T_0”
T_0
DenseData{...}
“contiguous block of bytes”

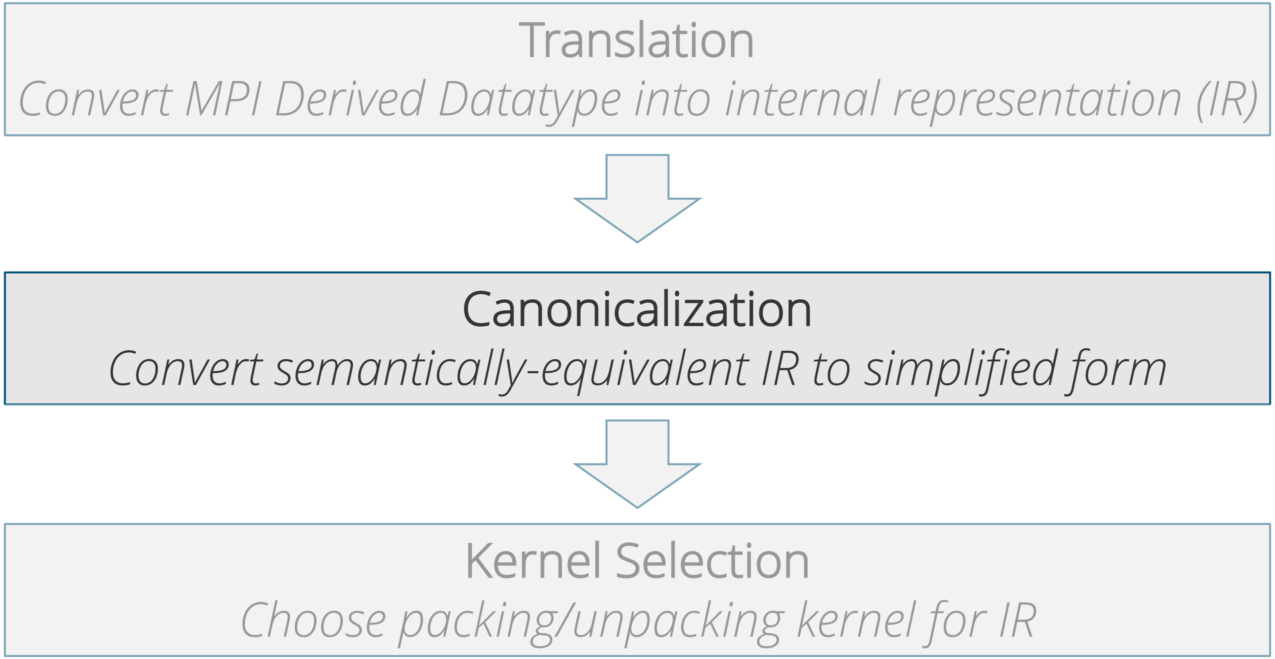
Example





StreamData{offset:0, count:E_2, stride:A_1*A_0}
StreamData{offset:0, count:1, stride:E_1*A_0}
StreamData{offset:0, count:E_1, stride:A_0}
StreamData{offset:0, count:1, stride:E_0}
StreamData{offset:0, count:E_0, stride:1}
StreamData{offset:0, count:1, stride:1}
DenseData{offset:0, extent: 1}

TEMPI Datatype Handling



Translation

Convert MPI Derived Datatype into internal representation (IR)

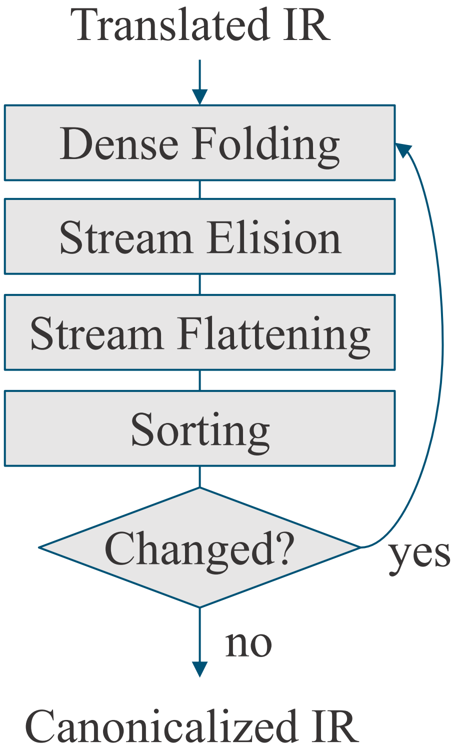
Canonicalization

Convert semantically-equivalent IR to simplified form

Kernel Selection

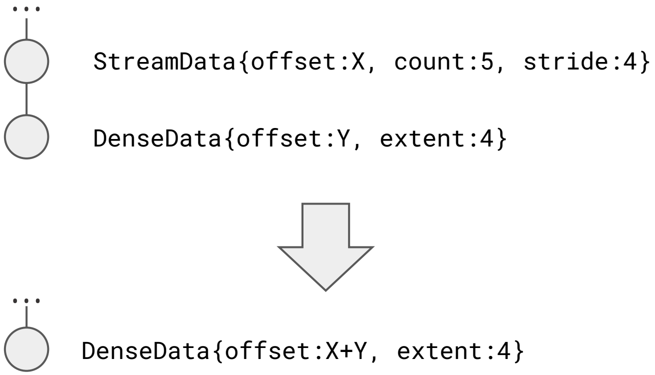
Choose packing/unpacking kernel for IR

Canonicalization

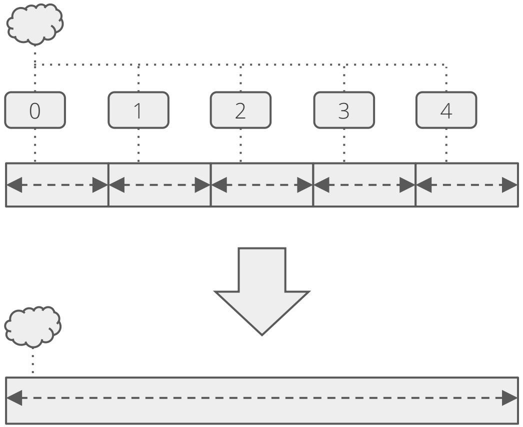


Canonicalization: Dense Folding

- `StreamData` is contiguous `DenseData` (parent of MPI named type)

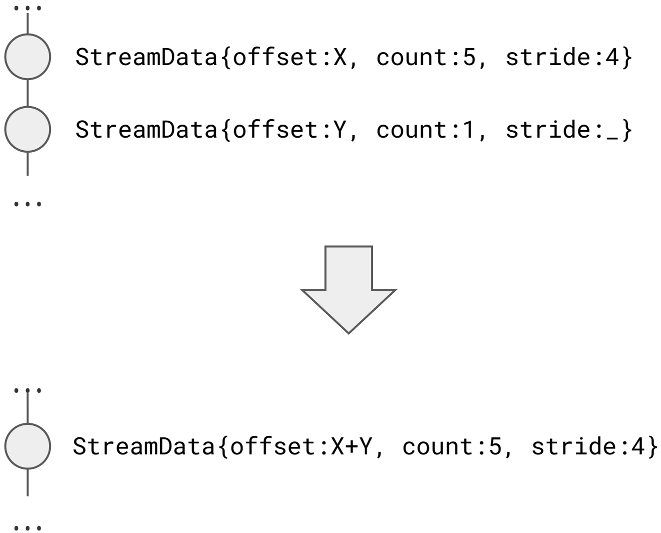


```
...
StreamData{offset:X, count:5, stride:4}
DenseData{offset:Y, extent:4}

...
DenseData{offset:X+Y, extent:4}
```



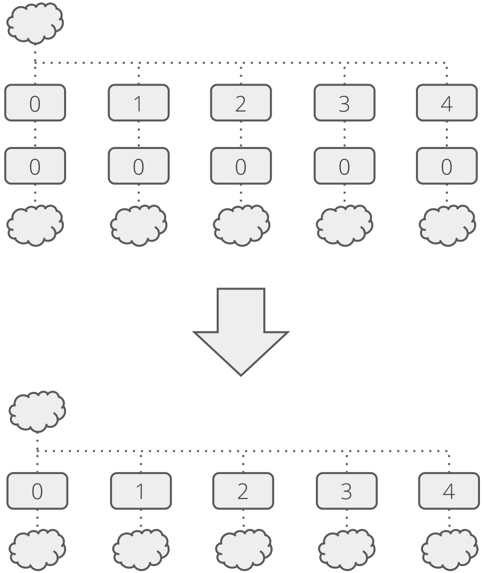
Canonicalization: Stream Elision

- Remove `StreamData` with `count = 1` (MPI Vector blocks commonly have one element)



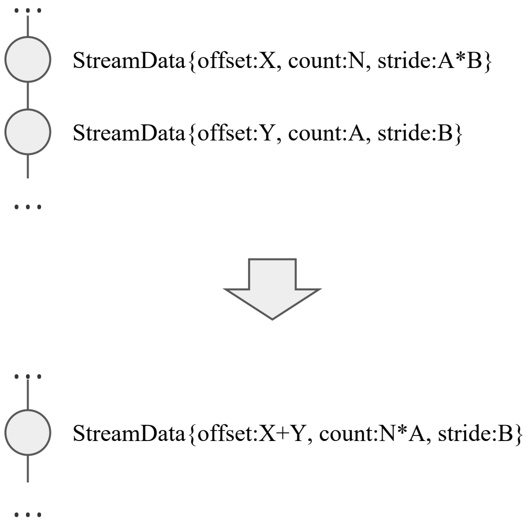
```
StreamData{offset:X, count:5, stride:4}
StreamData{offset:Y, count:1, stride:_}
```

```
StreamData{offset:X+Y, count:5, stride:4}
```



Canonicalization: Stream Flattening

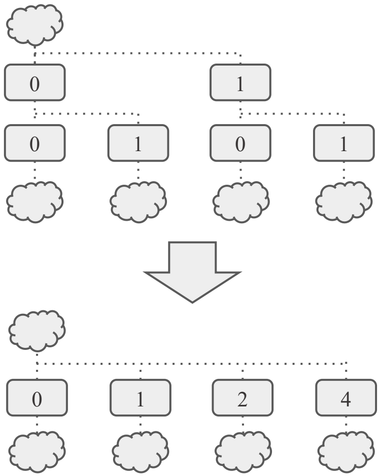
- e.g. two vectors of three vs. one vector of six



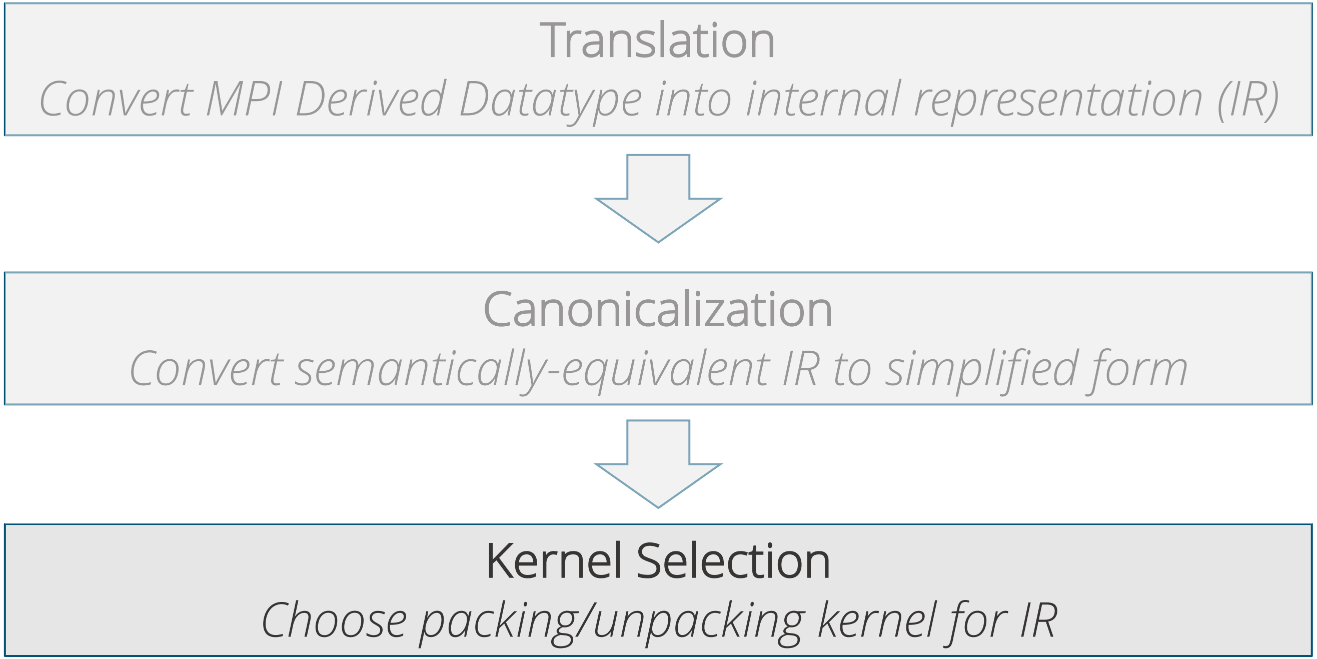
StreamData{offset:X, count:N, stride:A*B}

StreamData{offset:Y, count:A, stride:B}

StreamData{offset:X+Y, count:N*A, stride:B}



TEMPI Datatype Handling



Translation

Convert MPI Derived Datatype into internal representation (IR)

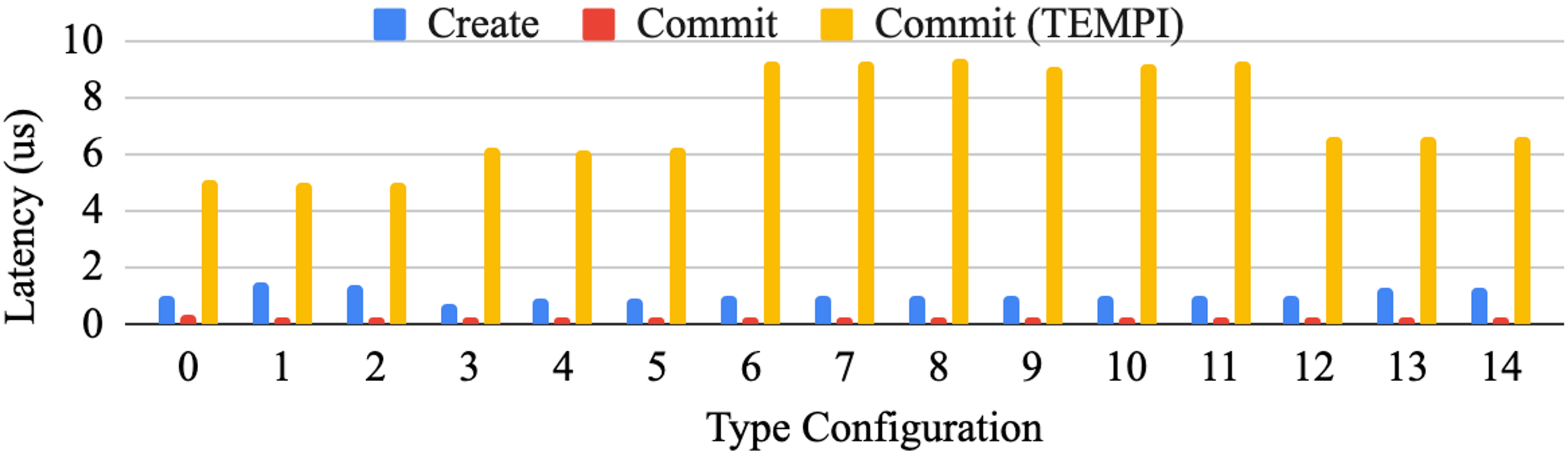
Canonicalization

Convert semantically-equivalent IR to simplified form

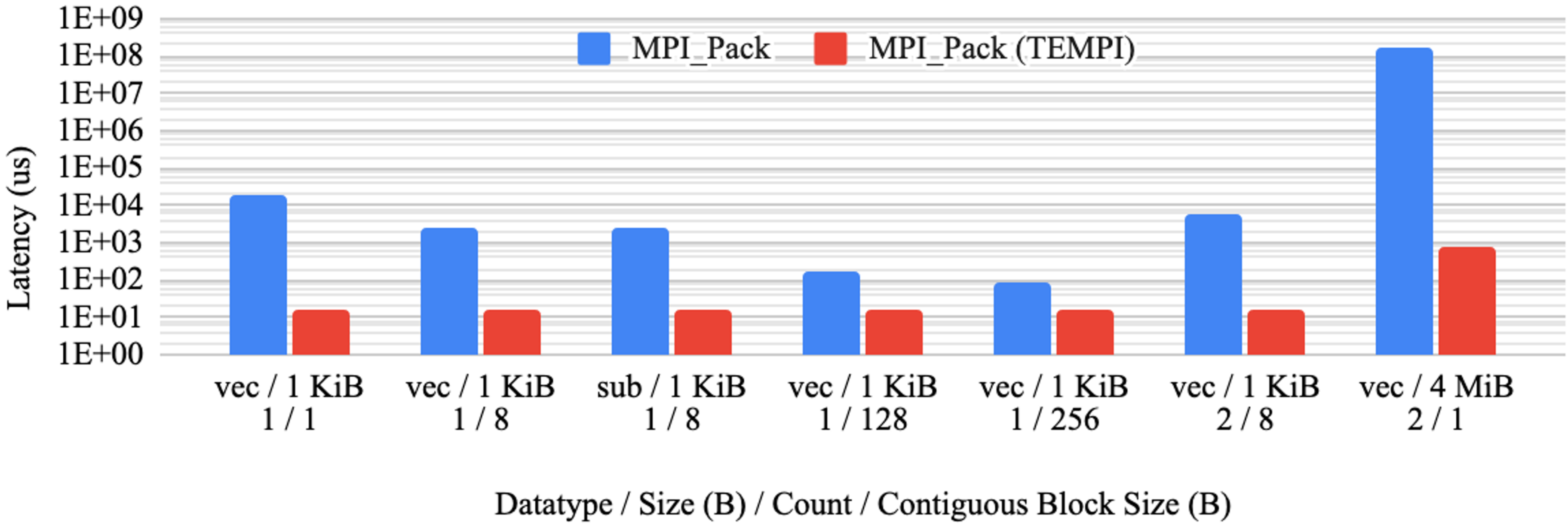
Kernel Selection

Choose packing/unpacking kernel for IR

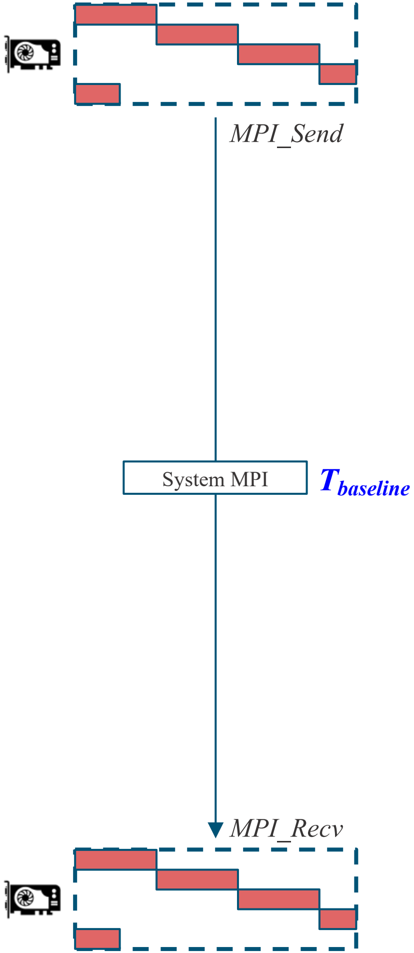
MPI_Type_commit Time



MPI_Pack Results

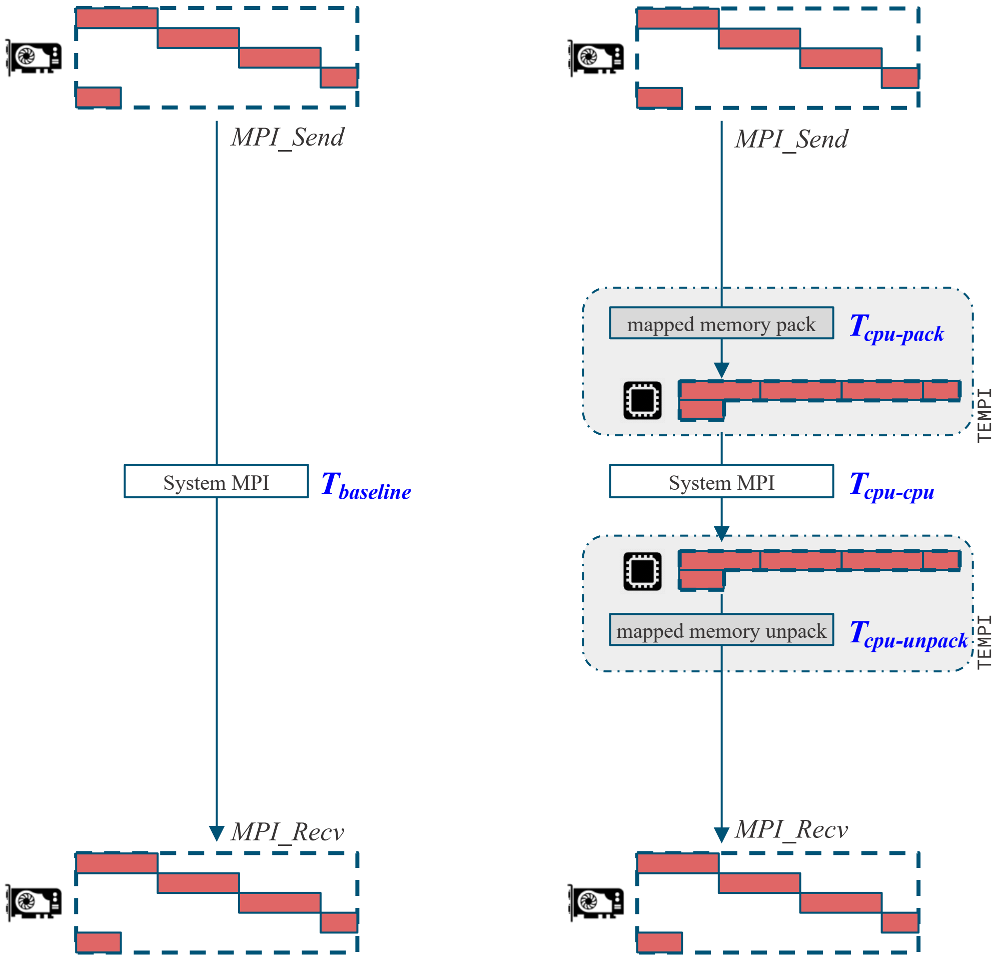


CUDA-Aware System MPI



CUDA-Aware System MPI

TEMPI “One-shot”



MPI_Send

System MPI $T_{baseline}$

MPI_Recv

MPI_Send

mapped memory pack $T_{cpu\text{-}pack}$

TEMPI

System MPI $T_{cpu\text{-}cpu}$

mapped memory unpack $T_{cpu\text{-}unpack}$

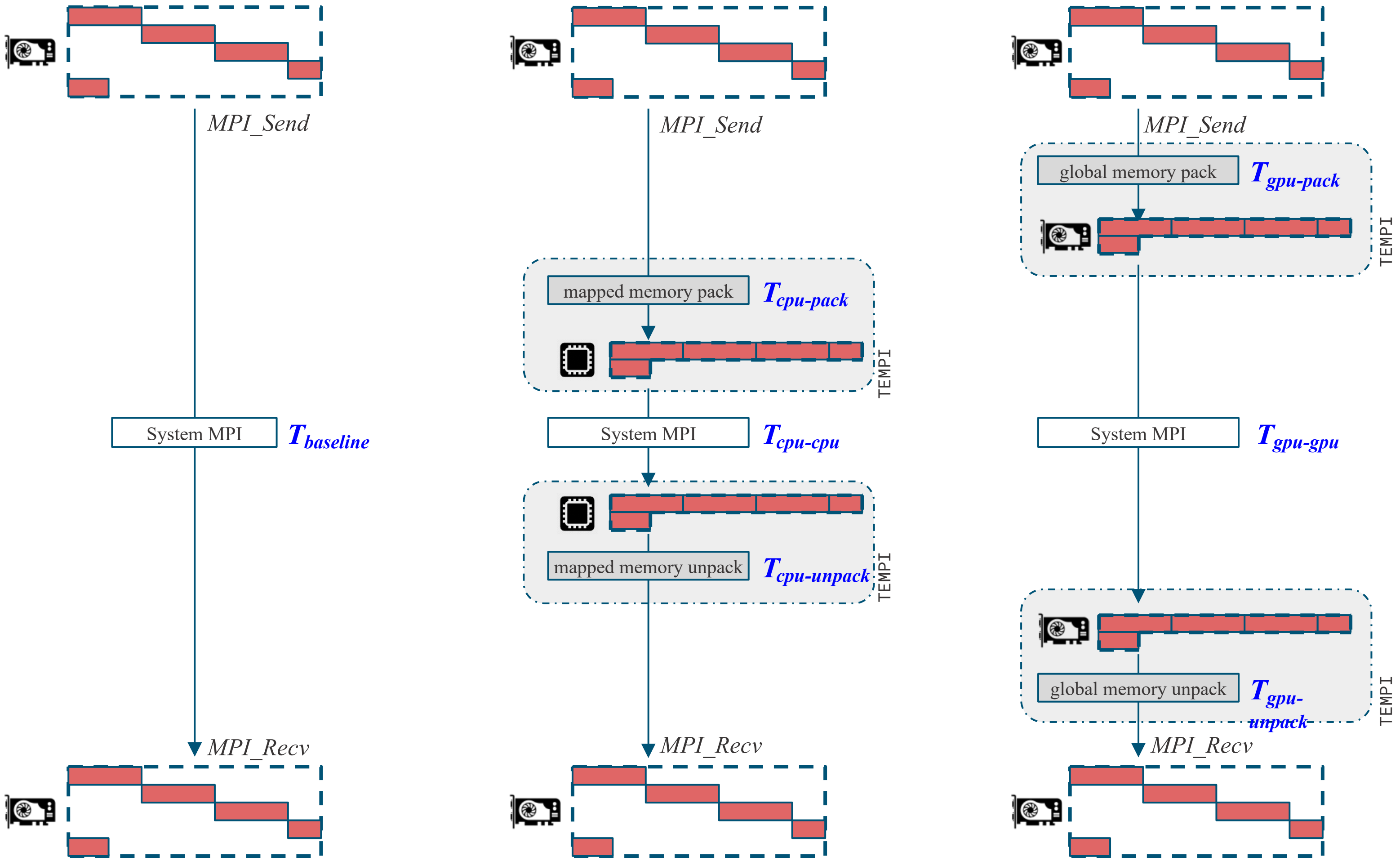
TEMPI

MPI_Recv

CUDA-Aware System MPI

TEMPI "One-shot"

TEMPI "Device"



MPI_Send

System MPI $T_{baseline}$

MPI_Recv

MPI_Send

mapped memory pack $T_{cpu-pack}$

TEMPI

System MPI $T_{cpu-cpu}$

mapped memory unpack $T_{cpu-unpack}$

TEMPI

MPI_Recv

MPI_Send

global memory pack $T_{gpu-pack}$

TEMPI

System MPI $T_{gpu-gpu}$

global memory unpack $T_{gpu-unpack}$

TEMPI

MPI_Recv

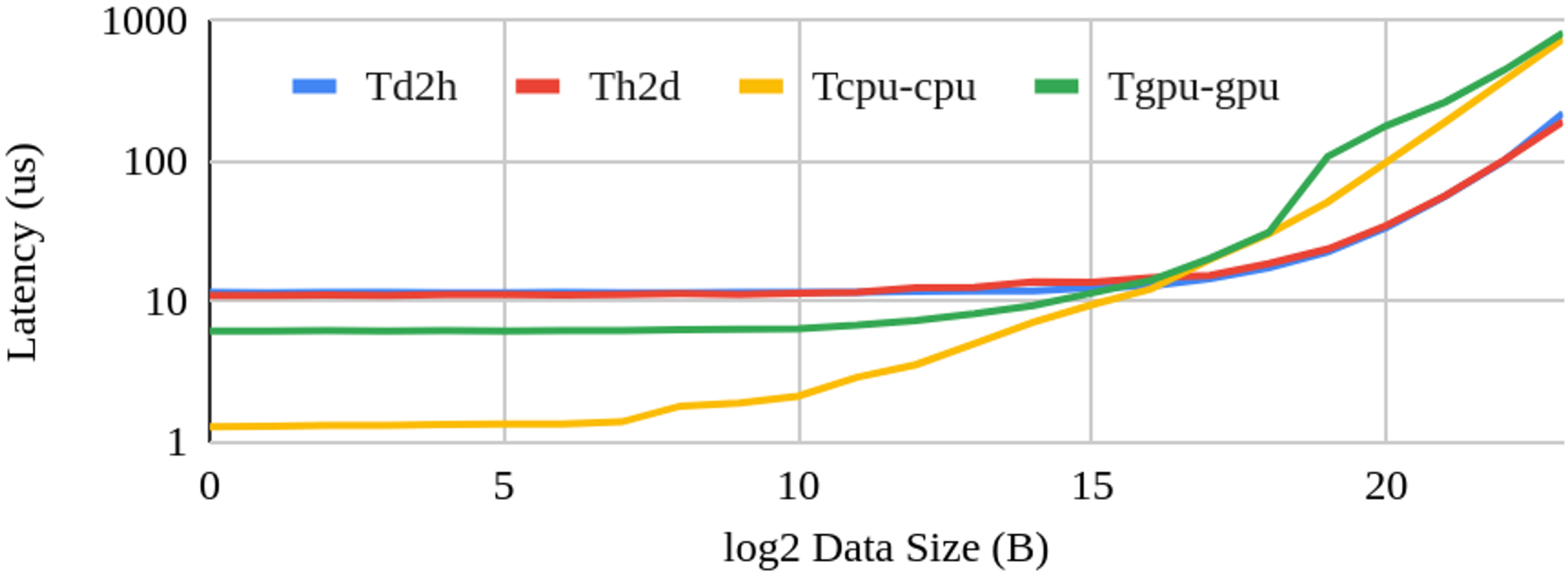
Performance Modeling

$$T_{device} = T_{gpu-pack} + T_{gpu-gpu} + T_{gpu-unpack}$$

$$T_{oneshot} = T_{host-pack} + T_{cpu-cpu} + T_{host-unpack}$$

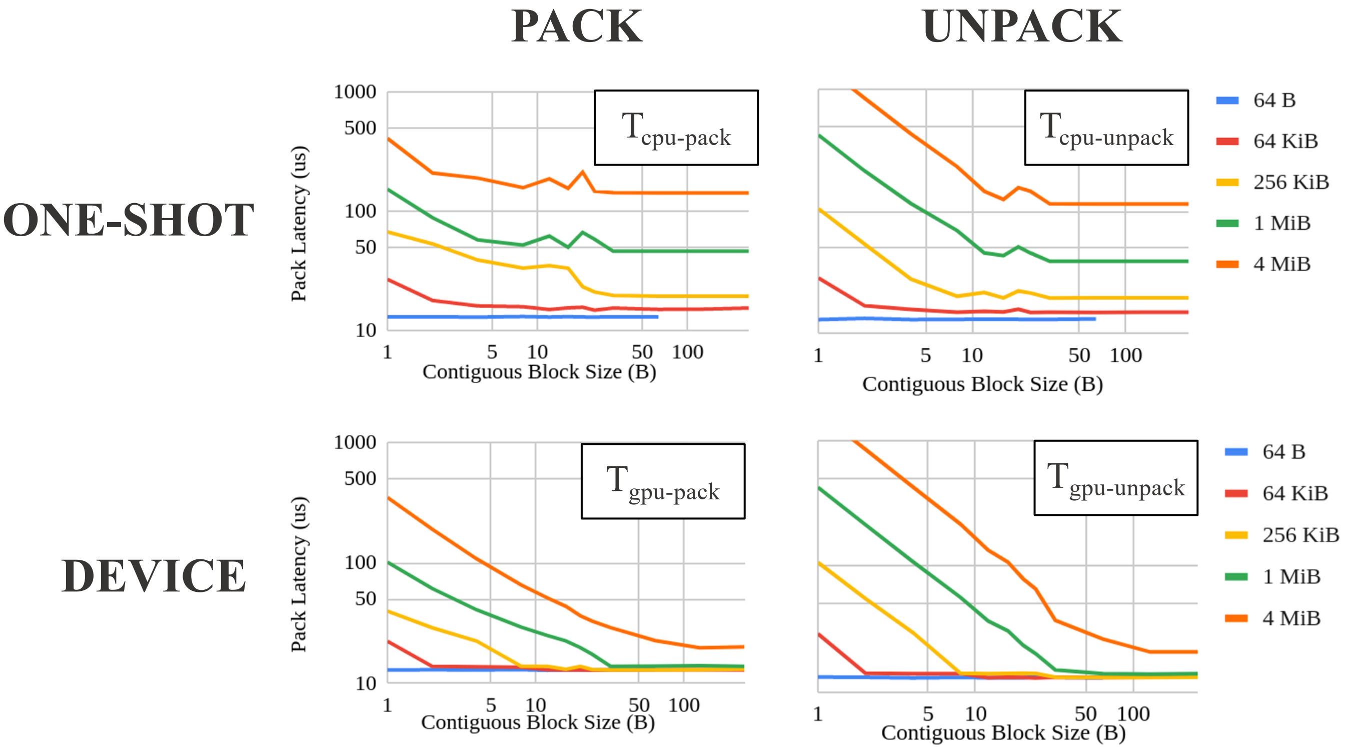
$$T_{staged} = T_{gpu-pack} + T_{d2h} + T_{cpu-cpu} + T_{h2d} + T_{gpu-unpack}$$

Contiguous Transfer Measurements



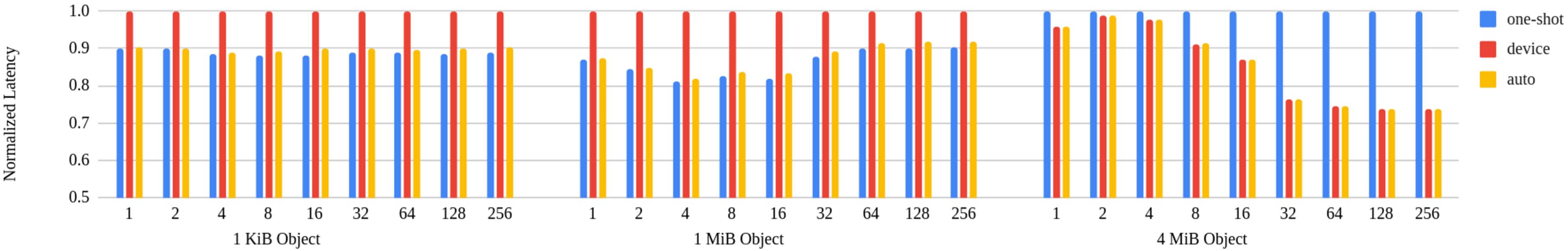
T depends on # bytes transferred

One-shot and Device Pack and Unpack



T depends on object size and block size

MPI_Send Microbenchmark

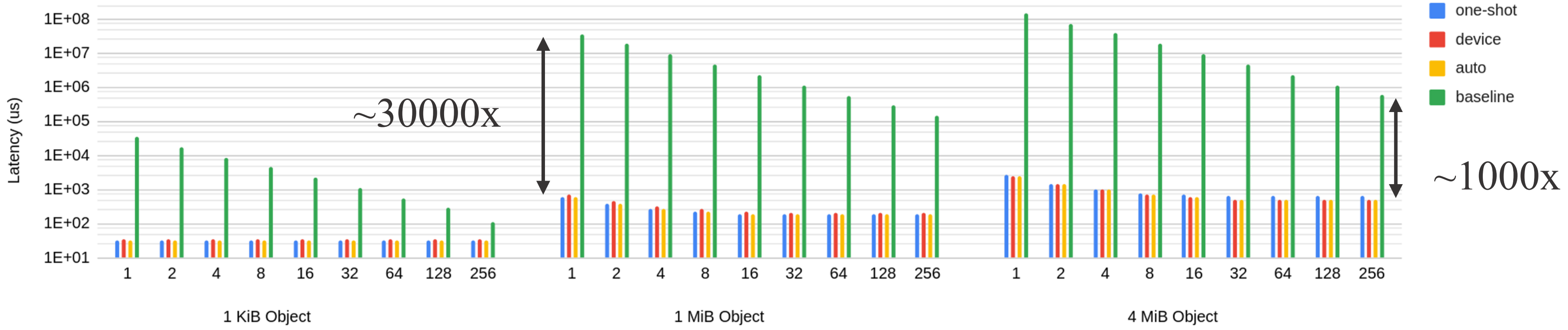


Minimal runtime performance-modeling overhead

Performance model reliably chooses faster method

MPI_Send / MPI_Recv

MPI_Send/Recv Latency for 2D objects with different block sizes



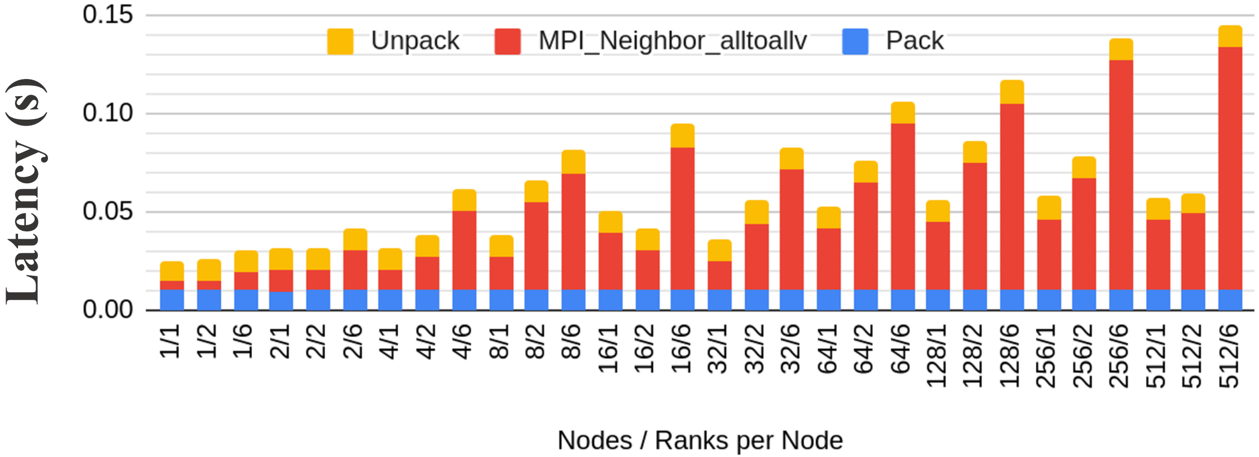
~30000x

~1000x

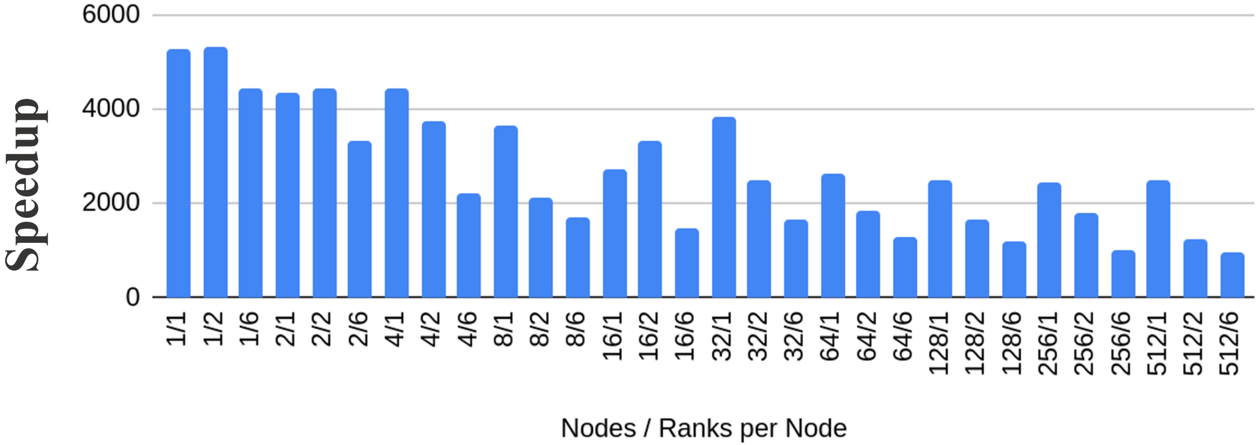
Large latency improvement from datatype handling
Small additional improvement from automatic method selection

Halo Exchange

With TEMPI, non-contiguous packing no longer dominates



Large speedup in 3D halo exchange

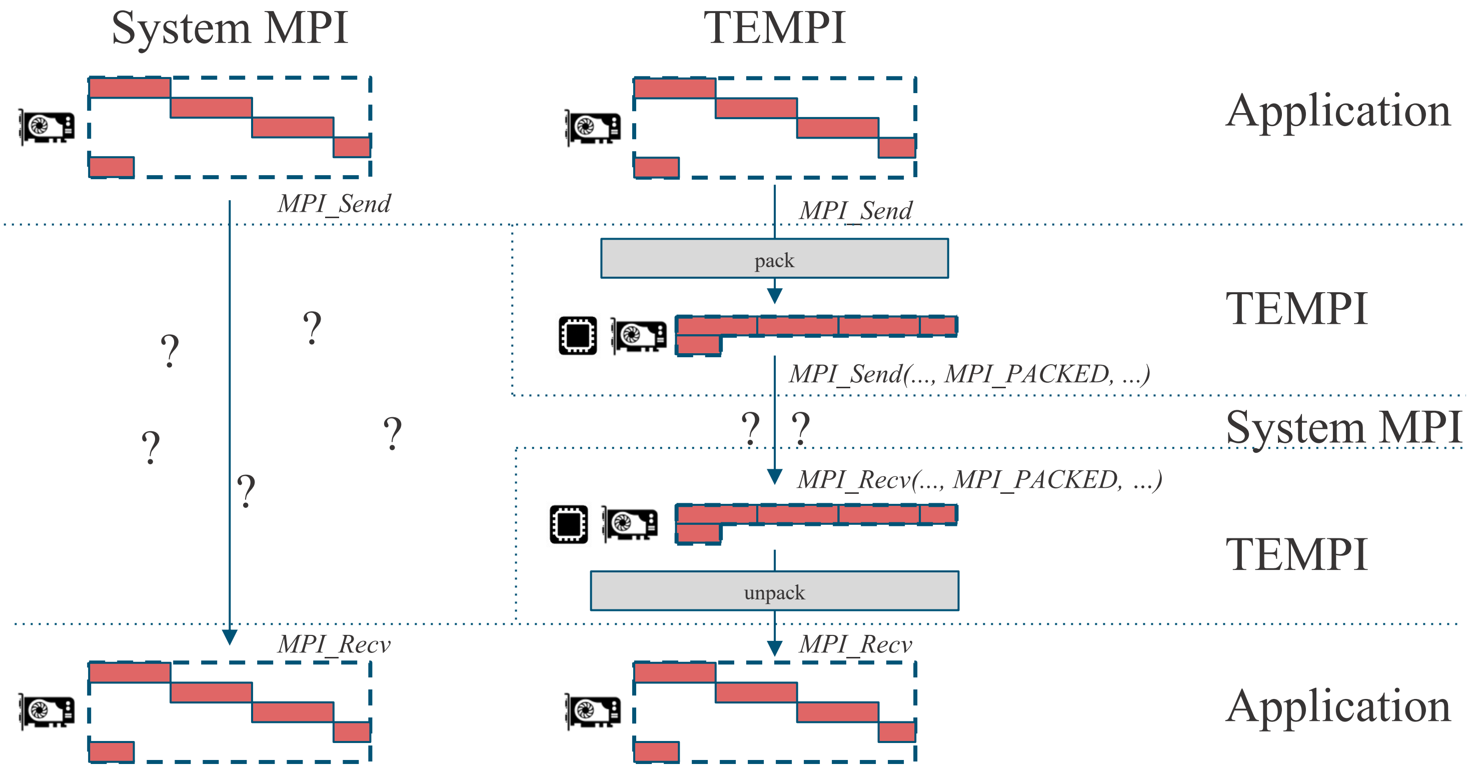


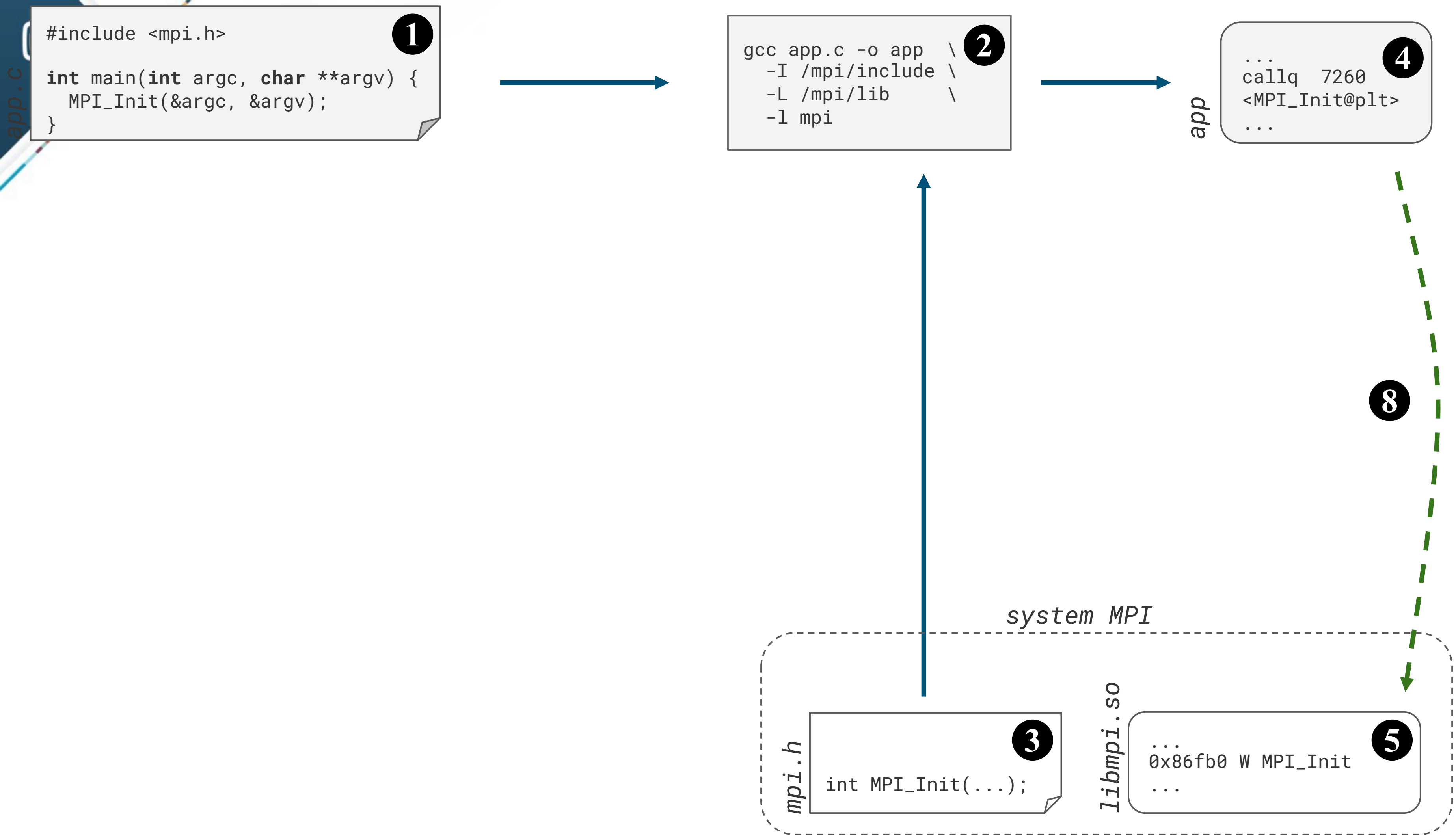
A Practical Challenge: Deploying MPI experiments on a Supercomputer

- Large-scale systems are tightly controlled
    - Can’t just make whatever changes you want
- Usually one MPI (or maybe two) are deployed on the system
    - Rarely bugfixed (if ever)
    - Even more rarely are new features added
- Difficult or impossible to make experimental modifications

- MPI has a well-defined standard
    - Take advantage of this + how OS loads libraries to inject modifications

TEMPI’s Architecture: Interposer Library





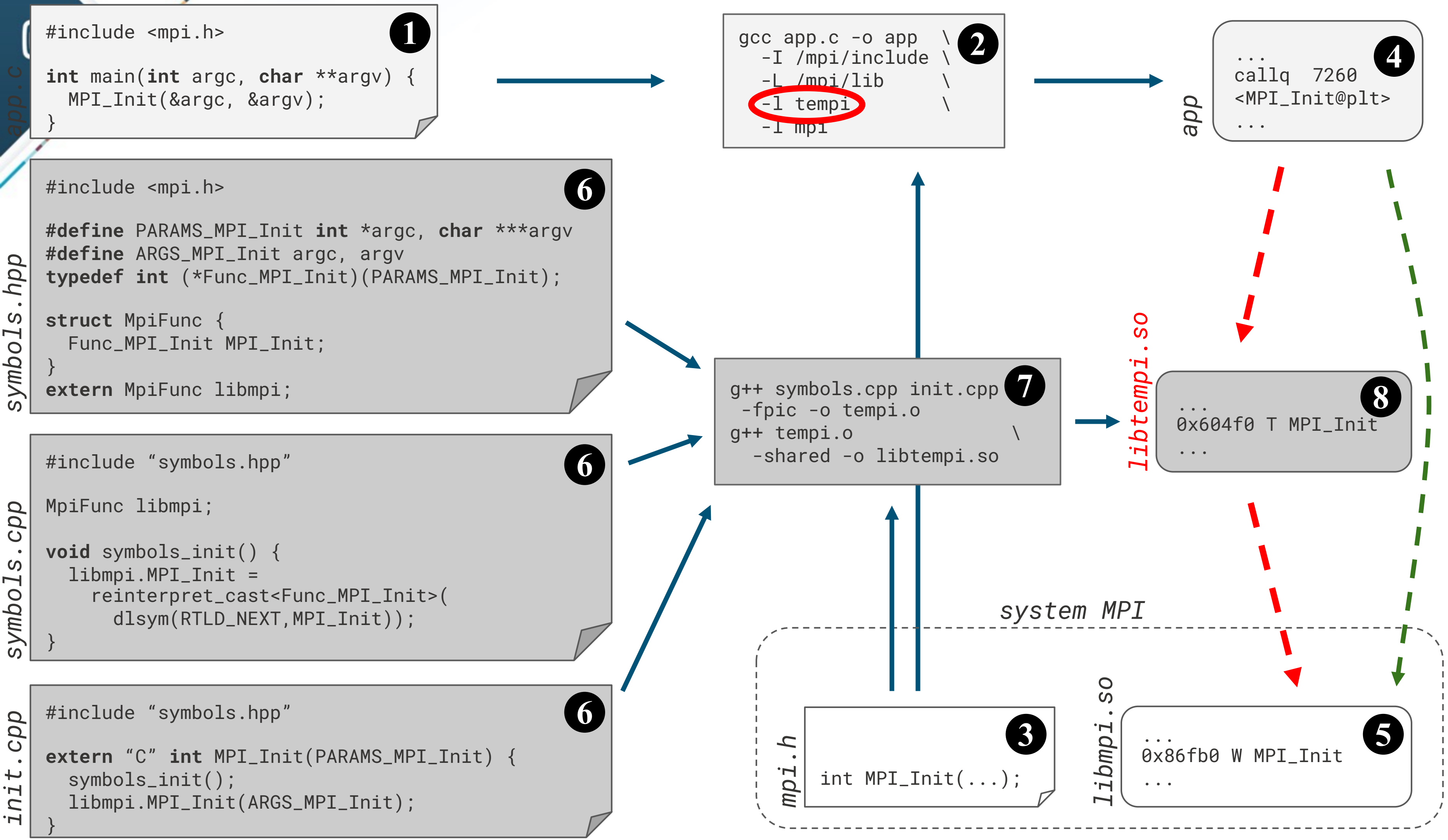
```
#include <mpi.h>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
}
```

```
gcc app.c -o app   \
  -I /mpi/include \
  -L /mpi/lib     \
  -l mpi
```

```
...
callq  7260
<MPI_Init@plt>
...
```

```
int MPI_Init(...);
```

```
...
0x86fb0 W MPI_Init
...
```



app.c

```
#include <mpi.h>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
}
```

```
gcc app.c -o app   \
  -I /mpi/include \
  -L /mpi/lib     \
  -l tempi        \
  -l mpi
```

app

```
...
callq  7260
<MPI_Init@plt>
...
```

symbols.hpp

```
#include <mpi.h>

#define PARAMS_MPI_Init int *argc, char ***argv
#define ARGS_MPI_Init argc, argv
typedef int (*Func_MPI_Init)(PARAMS_MPI_Init);

struct MpiFunc {
  Func_MPI_Init MPI_Init;
}
extern MpiFunc libmpi;
```

symbols.cpp

```
#include "symbols.hpp"

MpiFunc libmpi;

void symbols_init() {
  libmpi.MPI_Init =
    reinterpret_cast<Func_MPI_Init>(
      dlsym(RTLD_NEXT,MPI_Init));
}
```

init.cpp

```
#include "symbols.hpp"

extern "C" int MPI_Init(PARAMS_MPI_Init) {
  symbols_init();
  libmpi.MPI_Init(ARGS_MPI_Init);
}
```

```
g++ symbols.cpp init.cpp
 -fpic -o tempi.o
g++ tempi.o                 \
  -shared -o libtempi.so
```

libtempi.so

```
...
0x604f0 T MPI_Init
...
```

mpi.h

```
int MPI_Init(...);
```

libmpi.so

```
...
0x86fb0 W MPI_Init
...
```

Conclusions & Future Work

- Canonicalization approach works
  - 👍 Speedup for unmodified applications on OLCF summit
  - 👍 Any strided datatype
- Simple performance model to select GPU data transfer method
  - 👍 speedup
  - 👎 < 2x speedup
- Interposed library approach nice for experiments & prototype deployment
  - 👍 Easy to use without system privileges
  - 👎 Limited integration with existing MPI
- OpenMPI, MPICH, MVAPICH have other strategies
  - Use if available - covers some or all of the same problems
  - Comparison of MPI datatype performance would be useful to community
- github.com/cwpearson/tempi

Acknowledgements

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizon Network.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy Contract No. DE-AC05-00OR22725.

This work utilizes resources supported by the National Science Foundation’s Major Research Instrumentation program, grant #1725729, as well as the University of Illinois at Urbana-Champaign

Thank you

cwpears@sandia.gov

github.com/cwpearson/tempi

This work was completed prior to joining Sandia National Labs.