SC Asia 2018 (28/3/2018)

MACC: An OpenACC Transpiler for Automatic Multi-GPU Use

Kazuaki Matsumura[1,2], Mitsuhisa Sato[3,4], Taisuke Boku[4]
Artur Podobas[1], Satoshi Matsuoka[1,2]

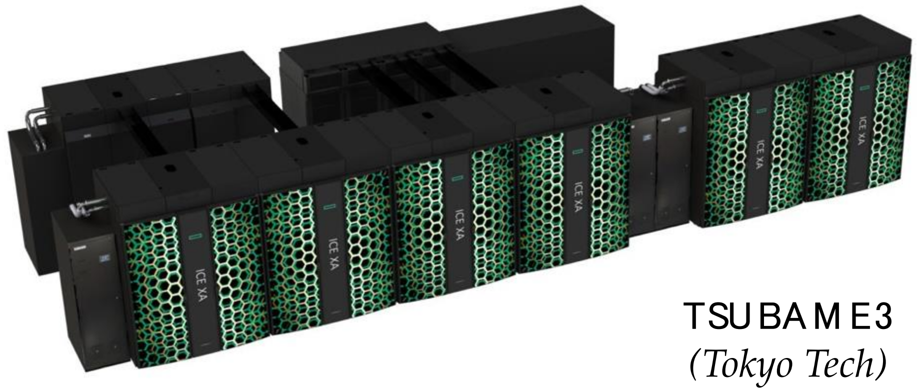
[1] Tokyo Institute of Technology [2] AIST-Tokyo Tech RWBC-OIL [3] RIKEN AICS [4] University of Tsukuba

Tokyo Tech

Introduction

Tokyo Tech

- Modern supercomputers are heterogeneously designed
  - Execution cooperatively with **<u>Accelerators</u>** (e.g. GPU, Intel MIC, FPGA)
- Several state-of-the-art supercomputers contain <u>multiple GPUs</u> **(<u>multi-GPU</u>)**



TSUBAME3
(Tokyo Tech)

- Using accelerators incurs additional programming cost
  - Through <u>primitives</u> (CUDA, OpenCL) or <u>abstract models</u> (DSL, Directive-based)

What is OpenACC ?

- Directive-based programming models complement naive code by putting **<u>directives</u>**
- OpenACC is a directive-based programming model developed by OpenACC organization
  - Supported by PGI Compiler (*NVIDIA*) and Cray Compiler, and experimentally by GCC
- Realizes <u>accelerator execution</u> by inserting directives into original C / Fortran source-code,

Example: N-Body Computation

Communication Code

GPU Invocation Code

CPU Code

```
cudaMalloc(&dev_m, N * sizeof(float));
cudaMalloc(&dev_p, N * sizeof(float3));
cudaMalloc(&dev_v, N * sizeof(float3));
cudaMemcpy(dev_m, m, N * sizeof(float),  cudaMemcpyHostToDevice);
cudaMemcpy(dev_p, p, N * sizeof(float3), cudaMemcpyHostToDevice);
cudaMemcpy(dev_v, v, N * sizeof(float3), cudaMemcpyHostToDevice);

for (int t = 0; t < TIME_STEP; t++) {
    kernel1<<<block_num, thread_num>>>(dev_m, dev_p, dev_v);
    kernel2<<<block_num, thread_num>>>(dev_m, dev_p, dev_v);
}
```

GPU Code

```
__global__ void kernel2(float *m, float3 *p, float3 *v){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int offset = tid * THREAD_SIZE;
    for (int j = 0; j < THREAD_SIZE; j++) {
        int i = offset + j;
        p[i].x += v[i].x * DT;
        p[i].y += v[i].y * DT;
        p[i].z += v[i].z * DT;
    }
}
```

NVIDIA CUDA

CPU + GPU Code

```
#pragma acc data copyin (p_x[N], p_y[N], p_z[N], m[N])
#pragma acc data copyout (v_x[N], v_y[N], v_z[N])
for (int t = 0; t < TIME_STEP; t++) {
#pragma acc parallel loop independent
    for (int i = 0; i < N; i++) { /* ... */ }

#pragma acc parallel loop independent
    for (int i = 0; i < N; i++) {
        p_x[i] += v_x[i] * DT;
        p_y[i] += v_y[i] * DT;
        p_z[i] += v_z[i] * DT;
    }
}
```

OpenACC Directives for Accelerators

Motivating example: Multi-GPU with OpenACC

Single-GPU Code

```
#pragma acc data\
  copyout(x[0:N]) present(y)
#pragma acc kernels
for (int i = 0; i < N; i++)
  x[i] = y[i] * y[i];
```

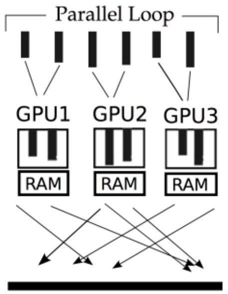
- + Concurrent Execution
- + Data Transfer
- + Loop Division

```
numgpus = acc_get_num_devices(DEVICE_TYPE);
#pragma omp parallel num_threads(numgpus)
{
  int tnum = omp_get_thread_num();
  int sz = N / numgpus;
  int lb = sz * tnum; int ub = lb + sz;
  acc_set_device_num(tnum, DEVICE_TYPE);
#pragma acc data copyout(x[lb:sz]) present(y)
#pragma acc kernels
  for (int i = lb; i < ub; i++)
  x[i] = y[i] * y[i];
}
```

Multi-GPU Code

- Manual efforts break out of the abstraction
- **We propose an automation method which requires no modification of original OpenACC source-code**

Related Work

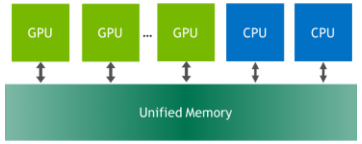


Quoted figure: 1)

"Integrating multi-GPU execution in an OpenACC compiler" (ICPP'13)

- Keeps the coherence of array chunks: *Bulk Synchronous Parallel model*
- Programmer has to provide the chunk size and additional annotations to optimize communications

Unified Memory of NVIDIA GPUs



Quoted figure: 2)

- Keeps all data coherent without user intervention → The problem is overheads.

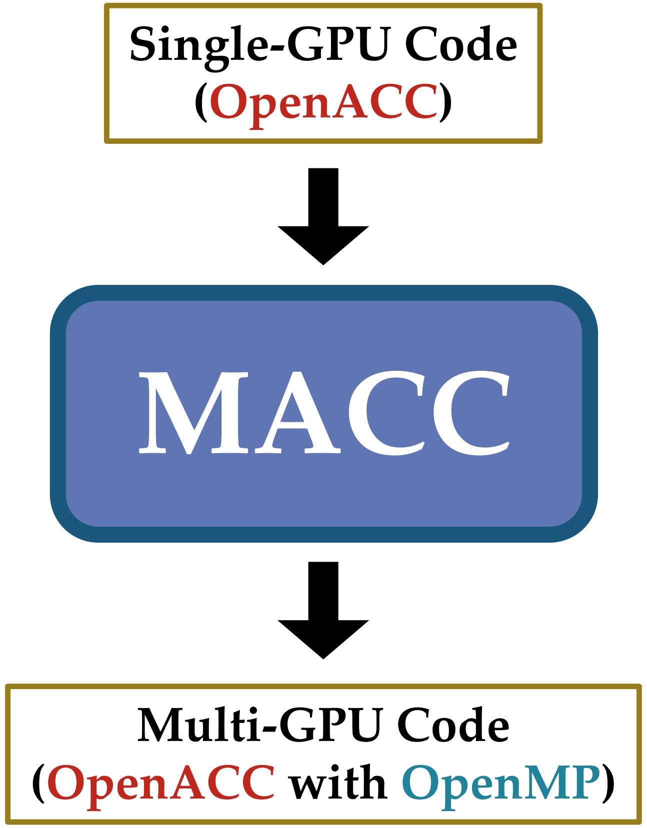
"Automatic data allocation and buffer management for multi-GPU machines" (TACO'13)

- Leverages the Polyhedral Model (a strict affine model) to detect fine dependencies

✗ `A[B[s]] = 1;`

1) Komoda Toshiya, Shinobu Miwa, Hiroshi Nakamura, and Naoya Maruyama. Integrating multi-GPU execution in an OpenACC compiler. In the 42nd International Conference on Parallel Processing (ICPP), 2013.
2) https://devblogs.nvidia.com/unified-memory-cuda-beginners/
3) Thejas Ramashekar, and Uday Bondhugula. Automatic data allocation and buffer management for multi-GPU machines. In ACM Transactions on Architecture and Code Optimization (TACO), Vol. 10, No. 4, Article 60, 2013.

Proposal: A Transpiler for Automatic Multi-GPU Use



- We propose a transpiler (source-to-source compiler) named **MACC**
- The output code can exploit multi-GPU without any manual effort, keeping original semantics and portability: **OpenACC** (GPU Parallel) + **OpenMP** (CPU Parallel)
- This proposal has a generality to support:
  - Multiple accelerators (not only GPUs)
  - Other directives for accelerators (e.g. **OpenMP**'s target directive)

Strategy: How to parallelize kernels over multi-GPU?

Tokyo Tech

Assumption 1. OpenACC primarily targets loop-level parallelism of the outermost loop

Assumption 2. Well-written OpenACC programs have continuous accesses

⟶ Equally divide the outermost loop for each GPU. If not dividable, execute on single-GPU.

Challenge: Inter-kernel communication

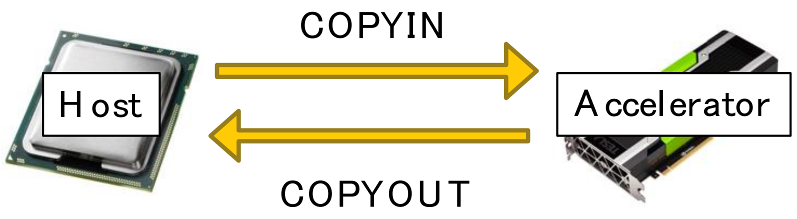
COPYIN

Inter-kernel

COPYOUT

```
#pragma acc data create(a[X:L])
{
#pragma acc parallel loop independent
    for (int i = 0; i < L; i++) { /* … */ }
-----------------------------------------
#pragma acc parallel loop independent
    for (int i = 0; i < L; i++) { /* … */ }
}
```

- Data-dependency among GPUs must be solved at the inter-kernel communication
- **We utilize upper/lower-bounds of array accesses to generate communications automatically**



MACC: Entire Process

Single-GPU Code (OpenACC)

Input →

← Output

Multi-GPU Code (OpenACC w/ OpenMP)

↓

OpenACC + OpenMP Compiler

↓

Multi-GPU Binary

1. Makes abbreviated notations flattened
   - #parallel loop→#parallel + #loop, #kernels copy(…)→#data copy(…) + #kernels
2. Replaces #kernels by using #parallel and #loop
   - We employed a basic loop-carried dependency checker
3. Iterative data-flow analysis for runtime detection of array regions
   - Collects array indexes, extracting variable representations
4. Converts #parallel, #data and #update, for each

MACC

```
#pragma acc parallel
{ /* … */ }
```

```
if (/* sections are changed */)
{ /* recalculate sections */ }
#pragma omp parallel num_threads(NUMGPUS)
{
  int tnum = omp_get_thread_num();
  set_gpu_num(tnum);
  set_data_section(/* ... */);
#pragma omp barrier
#pragma acc parallel
  { /* Splitted Loop */ }
}
```

```
#pragma acc data\
     copy(x[0:N])
{ /* ... */ }
```

```
#pragma omp parallel num_threads(NUMGPUS)
{
  copyin_routine(omp_get_thread_num(),x,0,N);
}
{ /* ... */ }
#pragma omp parallel num_threads(NUMGPUS)
{
  copyout_routine(omp_get_thread_num(),x);
}
```

```
#pragma acc update\
     host(x[a:b])
```

```
#pragma omp parallel num_threads(NUMGPUS)
{
  int tid = omp_get_thread_num();
  update_host_routine(tid, x, a, b);
}
```

- Convert to use MACC's communication routines
  - Which call OpenACC's routines
- **OpenMP** is for parallelizing multi-GPU execution
  - No restriction to be replaced by POSIX threads or **OpenACC**'s Async
  - But it benefits to:
    - Code generation (easy)
    - Better performance (than Async)
    - Multi-core execution (supported by just ignoring **OpenACC**)
    - Debugger/profiler use

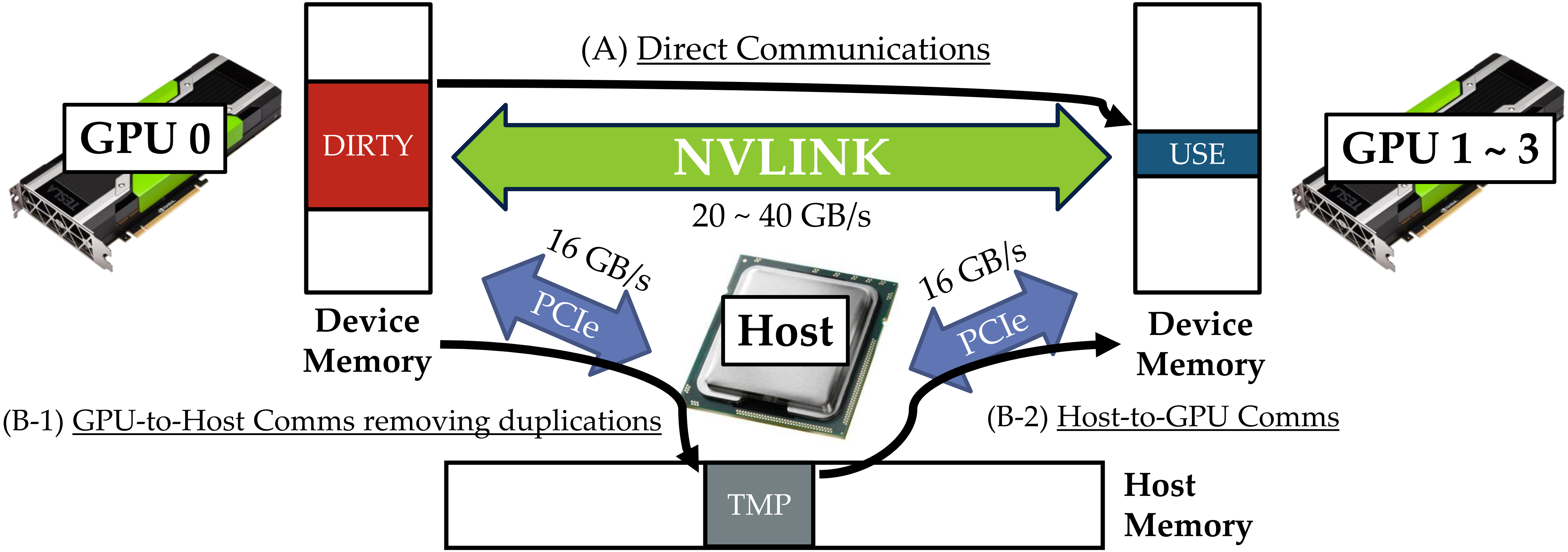
Tokyo Tech

MACC: Communication

- We leverage upper/lower-bounds of write/read accesses → ***USE***/***DEF*** *section*
  - Based on affine property, the sections are dynamically calculated for each combo of {GPU, Kernel, Array} by using upper/lower-bounds of loop-counters (MACC embeds runtime components)
  - Non-affine accesses (e.g. `A[B[s]] = 1;`) indicate the entire array

- MACC's routines manage updated regions for each combo of {GPU, Array} → ***DIRTY*** *section*
  - Before a kernel execution, necessary communications are deduced by the superposition of sections
  - Communications are executed through host memory removing duplicated transfers, or via NVLink

MACC: Communication



- Multi-GPU execution is enabled when the kernel's all DEF sections don't overlap among GPUs
  - The switch between single/multi-GPU execution is performed at runtime involving communications

MACC's Extension: Polyhedral Compilation

Tokyo Tech

- To complement our analysis, we employ PLUTO to perform loop-fission before transpilation

```
#pragma acc parallel loop
for (j1 = 0; j1 < M; j1++)
#pragma acc loop
  for (j2 = j1; j2 < M; j2++) {
    symmat[j1][j2] = 0.0;
    for (i = 0; i < N; i++)
      sysmat[j1][j2] +=
        data[i][i1] * data[i][i2];
    sysmat[j2][j1] = sysmat[j1][j2];
  }
```

Try

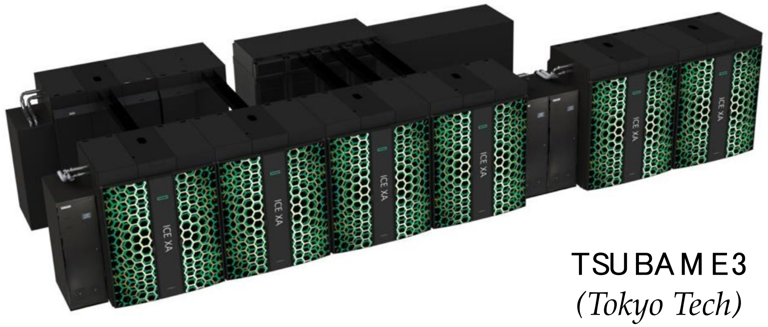
PLUTO (--no-fuse)

Reshape & Add #kernels

```
#pragma acc kernels
{
 for (j1 = 0; j1 < M; j1++)
    for (j2 = j1; j2 < M; j2++) { /*...*/ }
 for (j1 = 0; j1 < M; j1++)
    for (j2 = j1; j2 < M; j2++) { /*...*/ }
 for (j1 = 0; j1 < M; j1++)
    for (j2 = j1; j2 < M; j2++) { /*...*/ }
}
```

Implementation & Evaluation

- We implemented MACC's prototype as a converter of XcodeML/C (which represents C code in XML)
  - Currently multi-dimensional arrays are treated as 1-dimentional arrays
- We measured performances of several benchmarks written with OpenACC on one node of TSUBAME3
  - Comparing to MPI+OpenACC version and Unified Memory version

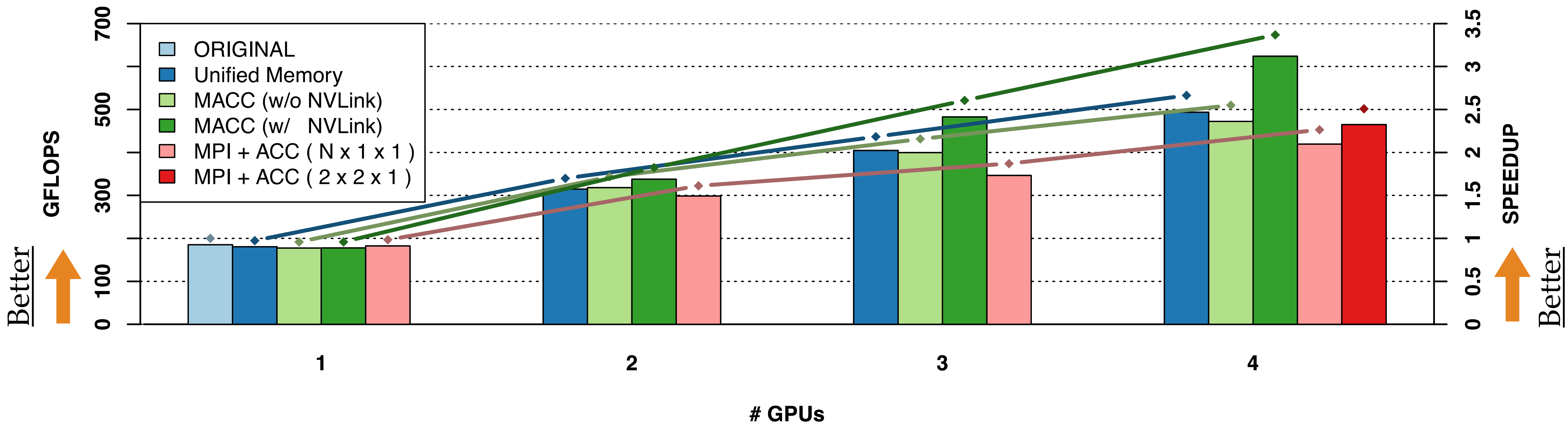


TSUBAME3
(Tokyo Tech)

CPU	Intel Xeon E5-2680 V4 (Broadwell-EP 14core) x 2
GPU	NVIDIA P100 (16GB HBM2@732GB/s) x 4
Compiler	PGI Compiler 17.10
CUDA	CUDA 9.0
NVLink	GPU0 ⇔ GPU2, GPU1 ⇔ GPU3: 40GB/s (one-way) Others: 20GB/s (one-way)

Himeno (Iterative 19-point Stencil Computation)

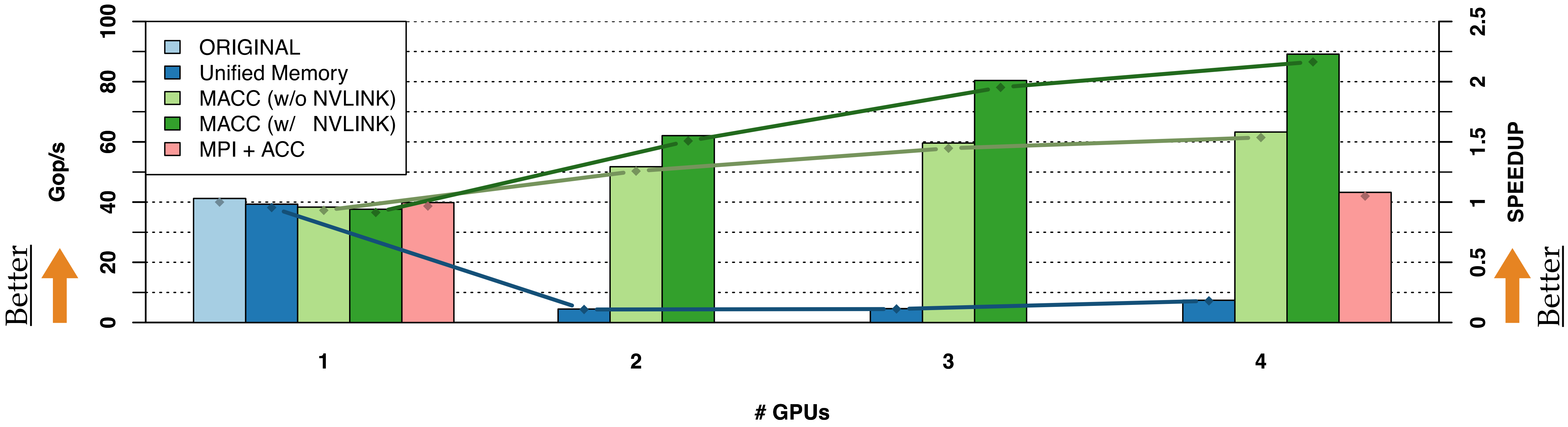
Size: $(i, j, k) = (256 \times 256 \times 512)$, Halo Communication (approx. $255 \times 511 \times 8$ bytes)



- MACC (w/ NVLink) achieved 3.36× speedup (32.1% performance increase compared to no NVLink)
- Unified Memory, that uses NVLINK, is slightly better than MACC (w/o NVLink)

NPB-CG (Iterative SpMV + Eigenvalue Calculation)

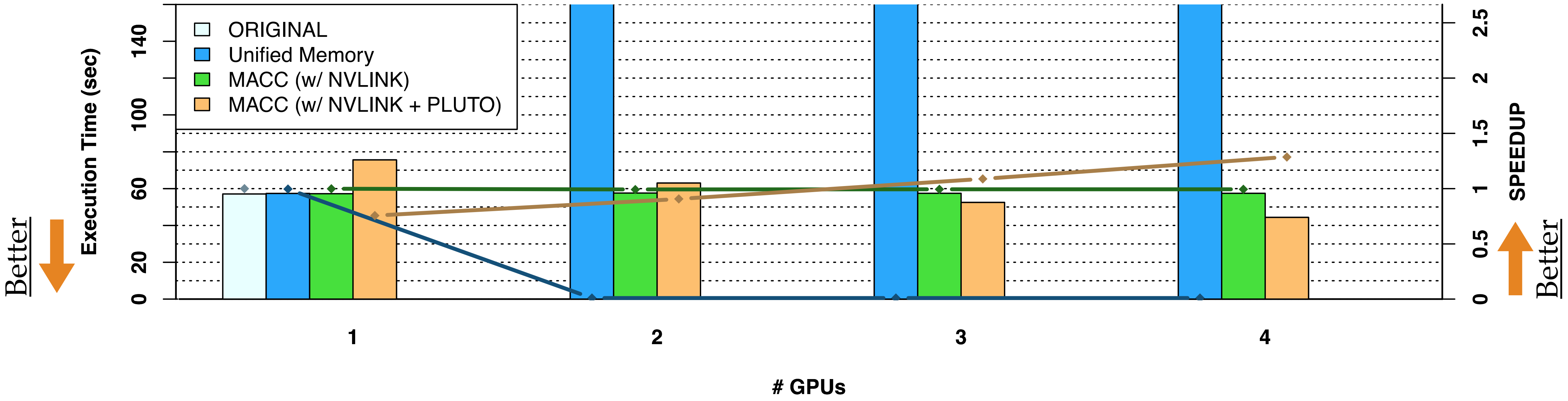
Size: rowsize = 150,000, All-to-All Communication (rowsize / GPUNUM × 8 bytes)



- MACC (w/ NVLink) gained the highest performance (40.9% performance increase compared to no NVLink)
- Unified Memory degraded the performance due to memory thrashing (frequent page fault & migration)
- MPI version (limited to proc=n^2) had low performances due to redundant communications

PolyBench/ACC COVAR (Covariance Matrix Calculation)

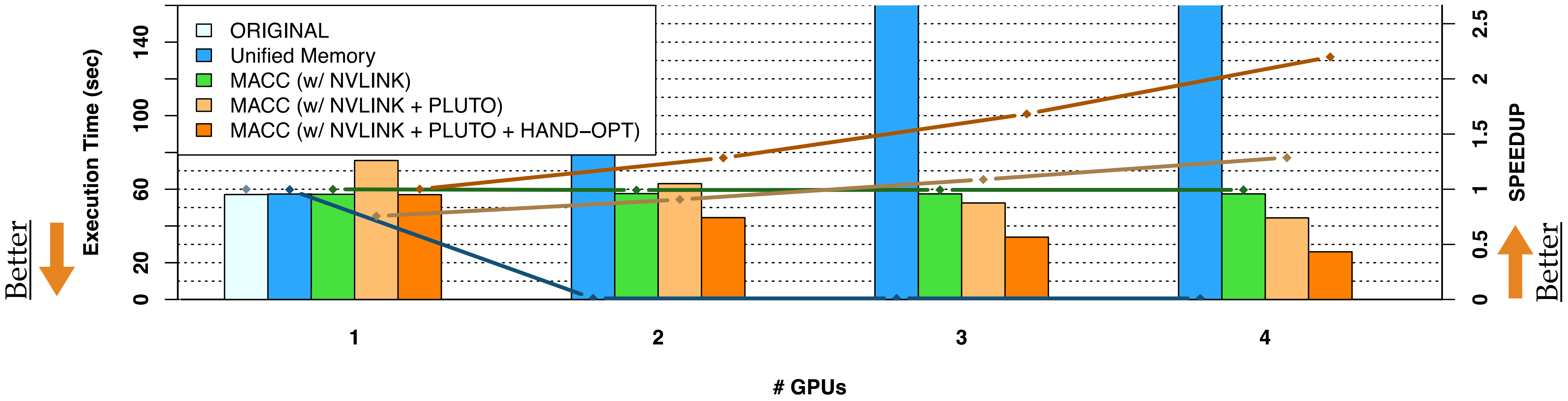
Size: 16,348 × 16,348



- A kernel combining symmetric-matrix creation (SC) + covariance calculation (CC) prevented multi-GPU exec
- After loop-fission by PLUTO, the SC and the CC were separated (1.29× speedup when using four GPUs)
  - The CC was executed on multi-GPU
  - The SC was executed **sequentially** ☹ , because of MACC and PGI's poor loop-carried dependency checker

PolyBench/ACC COVAR (Covariance Matrix Calculation)

Size: 16,348 × 16,348



- The SC was executed **sequentially** ☹ , because of MACC and PGI's poor loop-carried dependency checker
  - To solve the dependency, we added one directive line of loop construct into source-code after PLUTO
  - Still the SC was executed on single-GPU, but we achieved 2.20× speedup

    → We will need a loop model even in loop-dependency checker

Conclusion

- We built an OpenACC transpiler to use multi-GPU automatically
  - keeping the semantic and the portability
  - Communications are generated based on upper/lower-bounds of array accesses
- 3.36× speedup with stencil, and 2.16× speedup with NPB-CG when using four GPUs
- GPU-to-GPU communication via NVLink improved the performances
- Future work:
  - More analysis (affine and non-affine program analysis)
  - Work-sharing optimization (temporality, fine distribution)
  - Combining with task-based system
  - More accelerators, Hetero computing

Thank you

Back up

OpenACC (Execution Model)

- **`#pragma acc parallel`** : specifies a region executed on the accelerator (**<u>parallel region</u>**)

```
#pragma acc parallel
{ /* parallel region */ }
```

<u>Appendable clauses</u>:

```
num_gangs, num_workers, vector_length, if, async, reduction, …
```

- **`#pragma acc loop`** : describes the parallelism of a loop
  (automated, but still necessary for the performance improvement)

```
#pragma acc parallel num_gangs(8) num_workers(128) vector_length(128)
#pragma acc loop independent gang reduction (+ : sum)
for (int i = 0; i < N; i++)
#pragma acc loop independent worker
    for (int j = 0; j < N; j++)
#pragma acc loop independent vector
        for (int k = 0; k < N; k++) { /* ... */ }
```

<u>Appendable clauses</u>:

```
gang, worker, vector,
independent, seq, collapse,
reduction, …
```

architecture-independent parallelism

<u>Note: OpenACC provides abbreviated notations mixing several directives</u>

OpenACC (Execution Model)

Tokyo Tech

- **`#pragma acc kernels`** : treats loops as **kernels**, as far as possible
  - PGI Compiler mainly treats a **tightly nested loop** as a kernel

```
#pragma acc kernels
{
    for (int t = 0; t < MT; t++)
        for (int i = 0; i < MI; i++)
            a[i] += b[t] * a[i];
}
```

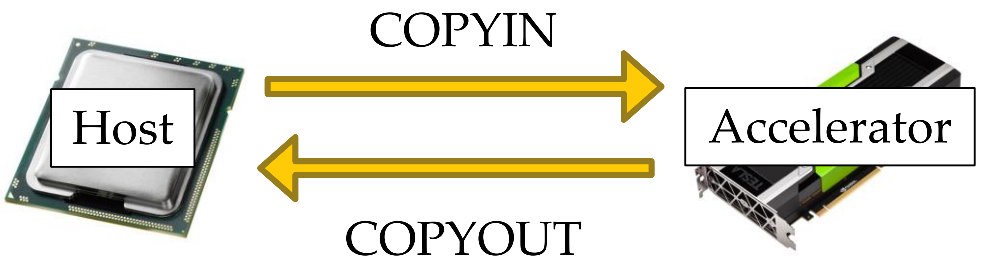
OpenACC Kernel: a nested loop executed in parallel on the accelerator

Tightly Nested Loop: nested loop which has just one statement inside except for the innermost

OpenACC (Memory Model)

- `#pragma acc data` : defines variables on the accelerator
  - A GPU manipulates its own memory (called **<u>device memory</u>**), so dependent data must be transferred
  - Before and after the region, communications corresponding specified clauses are occurred

```
#pragma acc data create(a[X:L]) copyin(b[X:L]) copyout(c[X:L]) copy(d[X:L]) present(e)
{
   /* Array 'a' 'b' 'c' 'd' and 'e' live here */
}
```



CREATE: Allocates memory space without transfers

COPY: COPYIN + COPYOUT

PRESENT: Already exists

- `#pragma acc update` : updates variables being defined on the accelerator (COPYIN or COPYOUT)

```
#pragma acc update host(a[start:length]) device(b[start:length])
```

MACC: Actual Example

Before

```
#pragma acc parallel loop gang reduction (+ : sum) present(a)
for (i = X; i < Y; i++) {
  sum += a[i + p];
}                                                   PARALLEL REGION
```

After

```
{
  static int sections_are_changed = 1;
  sections_are_changed =
    (sections_are_changed || last_p != p || last_X != X || last_Y != Y);

  if (sections_are_changed) {
    section_are_changed = 0; last_p = p; last_X = X; last_Y = Y;

    calc_loop_sections(loop_sections, X, Y,
                       1 /* increment */,
                       0 /* whether to execute when X==Y */);

    init_uses(a_uses, 1 /* affine */); init_defs(a_defs, 0 /* none */);
    for (i = 0; i < NUMGPUS; i++) {
      update_section(a_uses[i], loop_sections[i].lb + p);
      update_section(a_uses[i], loop_sections[i].ub + p);
    }

    if (is_overlapping(a_defs)) {
      /* reconstruct for single GPU execution */
    }
  }
}                                                   SECTION CALCULATION
#pragma omp parallel num_threads(NUMGPUS) reduction (+ : sum) private (i)
{
  int tnum = omp_get_thread_num();
  set_gpu_num(tnum);

  set_data_section(tnum, a, a_uses, a_defs);
#pragma omp barrier                                 COMMUNICATION

#pragma acc parallel present(a)
#pragma acc loop gang reduction (+ : sum)
  for(i = loop_sections[tnum].lb; i <= loop_sections[tnum].ub; i++) {
    sum += a[i + p];
  }                                                 PARALLEL REGION
}
```