JACC: An OpenACC Runtime Framework with Kernel-Level and Multi-GPU Parallelization

Kazuaki Matsumura, Simon Garcia de Gonzalo, Antonio J. Peña

Barcelona Supercomputing Center (BSC)

kazuaki.matsumura@bsc.es

18/12/21 @ HiPC 2021

The EPEEC project has received funding from the European Union's Horizon 2020 research and innovation programme under the grant agreement Nº 801051

Introduction

```
__global__ void kernel(
          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;
    }
}
```

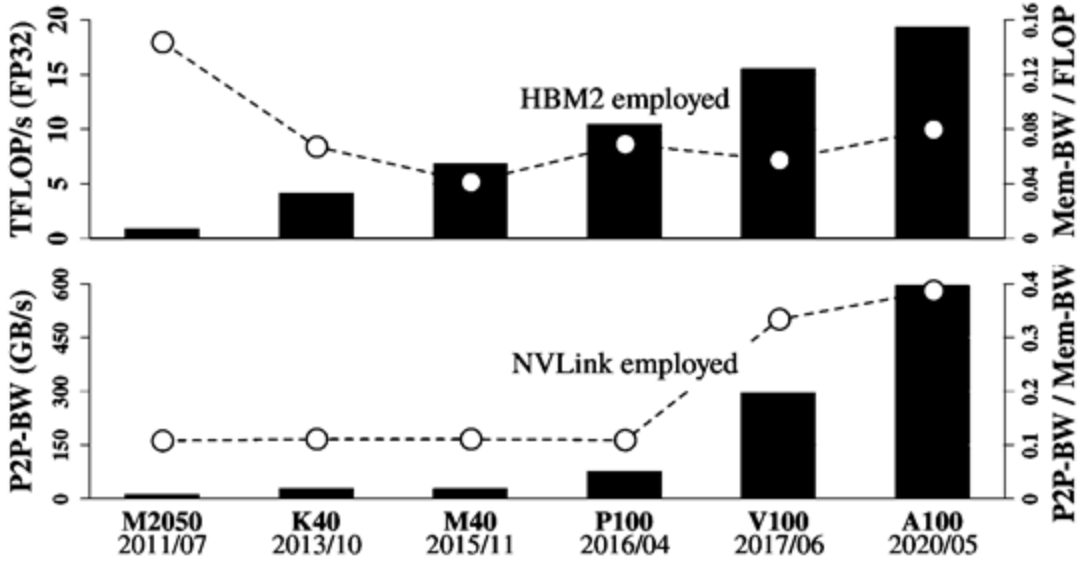

```
#pragma acc data copyin (p_x[N], p_y[N], p_z[N], m[N])
#pragma acc data copyout (v_x[N], v_y[N], v_z[N])
for (int t = 0; t < TIME_STEP; t++) {
#pragma acc parallel loop independent
    for (int i = 0; i < N; i++) { /* ... */ }

#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

- **Modern supercomputers often have heterogeneous designs**
  - Execution performed on CPUs & *Accelerators* (e.g. GPUs/FPGAs)
- **Utilizing accelerators poses additional programming cost**
  - Through primitives (CUDA/OpenCL) or *Abstract Models* (DSL/Directive)
  - The latter introduces less engineering efforts w/ limited interfaces → *Less Efficiency*

Introduction

- **Exploiting better performance on accelerators is not straightforward**
  - Higher compute performance, *Lower Bandwidth/FLOP Ratio*
  - Global/local memory access, synchronization, on-chip resource use and little parallelism might limit the performance
  - GPU's parallelism: instruction-level, thread-level, kernel-level or device-level
- **Currently, the NVLink interconnect is enhancing peer-to-peer communication among multi-GPU**
  - *Increased P2P-Bandwidth/Mem-Bandwidth*
  - This allows not only compute-intensive but memory-intensive applications to utilize several devices



OpenACC

```
#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

- OpenACC offers compiler directives to program accelerators in existing languages
- Without introducing vender-specific languages such as CUDA, users are allowed to parallelize their code and rely on the compiler for generating device-specific application code

OpenACC

- OpenACC = **kernels** + **routines**
  - An OpenACC kernel is the unit of program execution on accelerators launched with a specified parallelism
  - The environment for kernel execution can be controlled by OpenACC routines (e.g. data copies and sync behaviors)

```
acc_set_device_num(…);
acc_copyin(…);
```

- **OpenACC directives** are provided for specifying a portion of code as a kernel and defining data
  - Data-related directives can be replaced by routine calls
  - Kernel directives have to be embedded with original code for pre-execution compilation

```
#pragma acc data copyin(…)
#pragma acc kernels …
```

Optimizing Example: Multi-GPU Use

Config

```
for (int d = 0; d < NUM_DEV; d++) {
  acc_set_device_num(d , 0);
  int length = N / NUM_DEVICES;
  int init = length * d;
  int until = length * (d + 1);
```

Kernel

```
#pragma acc parallel loop\
      independent async(d)
  for (int i = init; i < until; i++) {
    /* ... */
  }
```

Comm

```
  for (int d2 = 0; d2 < NUM_DEV; d2++) {
    cudaMemcpyAsync( … );
  }
}
```

- **Each kernel needs code extension**
  - Loop splitting / Async / Communication
  - Many other factors such as runtime information are concerned in real applications

→ *In-situ kernel declarations bring much complexities*

- **Once kernels are declared, they do not allow any update**

→ *has to prepare adjusted kernels before the compilation*

Proposed Framework: JACC

- Our framework JACC is designed to serve as a **transparent layer between runtime and compiler**
  - All the components of OpenACC, here, are provided as runtime routines
  - Having the ability to perform dynamic analysis and runtime compilation, allowing further efforts at runtime
- We adapt several **automated optimization techniques** on JACC
  - Asynchronous execution / On-the-fly kernel optimization / Multi-GPU utilization

Execution Flow of JACC

Input Code

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

① Routine Use

JACC Code

```
/* Entry of #pragma acc data */
jacc_create(x, N * sizeof (float));

/* #pragma acc parallel loop */
jacc_kernel_push(
  "#pragma acc parallel present (x, y)\n"
  "#pragma acc loop\n"
  "for (int i=0; i<N; i++) /* ... */",
  /* args */ , /* flags */ );

/* Exit of #pragma acc data */
jacc_copyout(x, N * sizeof(float));
```

② Exec

Dynamic Code

```
void kernel0(float *x,
             float *y, size_t N) {
#pragma acc parallel present(x, y)
#pragma acc loop
for (int i=0; i<N; i++)
    x[i]=y[i] * y[i];
}
```

③ Generate Code

④ Compile (PGI/GCC)
Link
Caching Binary
Execution

JACC Runtime

Data Management
- Array Mapping
- CPU-to/from-Device Communication

Kernel Execution
- Code Gen/Compile
- Parameter Calculation
- Extended Execution

Automated Asynchronous Execution

Input Code

```
#pragma acc kernels create(a, b) copyout(c)
{ for (…) a[i] = i; for (…) b[i] = i;
  for (…) c[i] = a[i] + b[i];
  for (…) c[i] = b[i] + c[i];  }
```

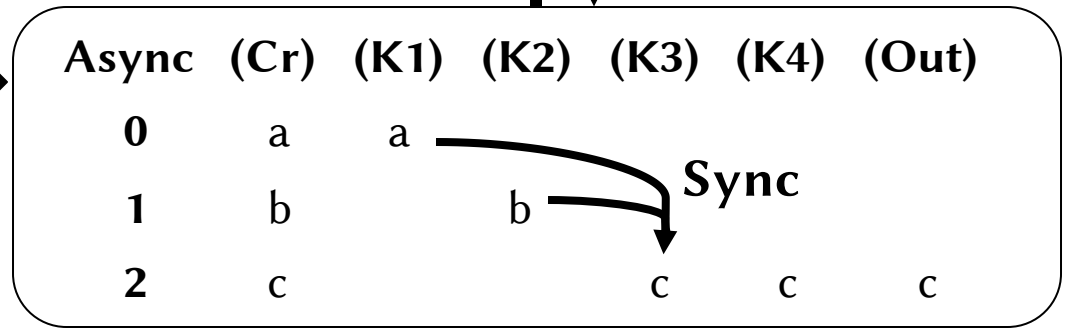
JACC Code

```
jacc_create(…);       // Create a/b/c

jacc_kernel_push(…); // Initialize a

jacc_kernel_push(…); // Initialize b

jacc_kernel_push(…); // Update c from a/b

jacc_kernel_push(…); // Update c from b/c

jacc_copyout(…);      // Copyout c; Del a/b
```

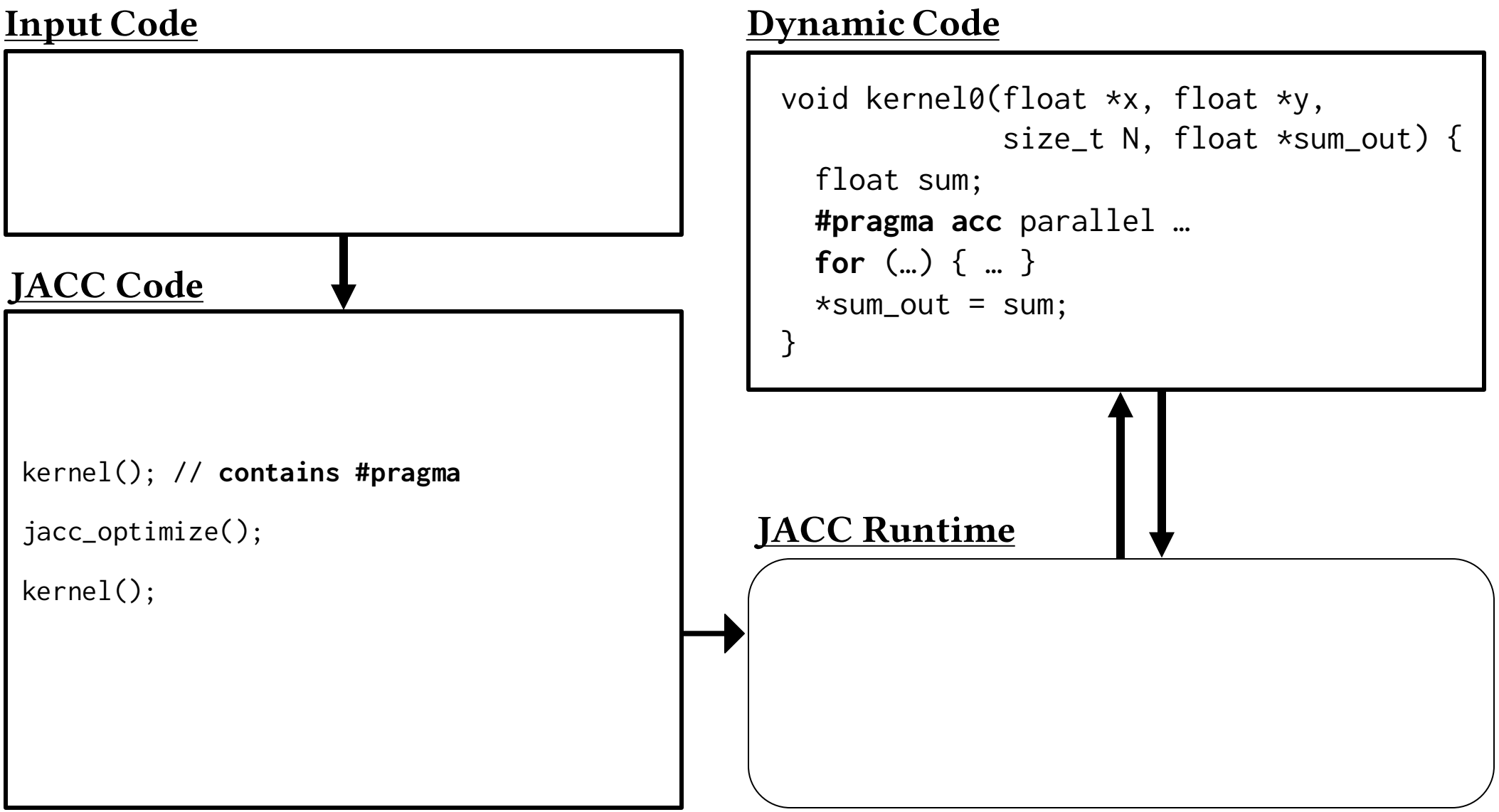
Dynamic Code

```
void kernel0(…, int async) {
// …
#pragma acc parallel … async(async)
  for (…) {
    a[i] = i;
  }
}
```

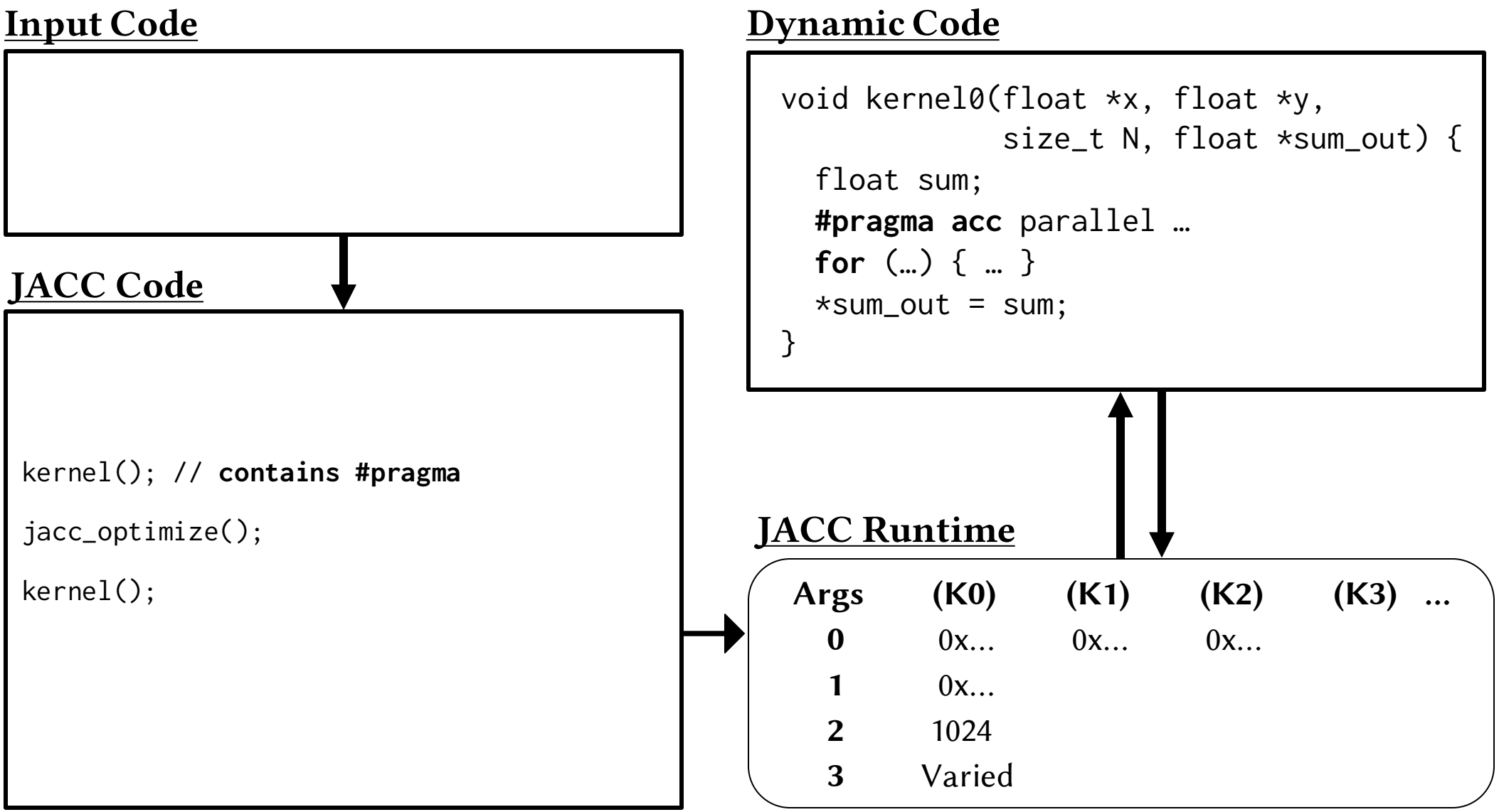
JACC Runtime



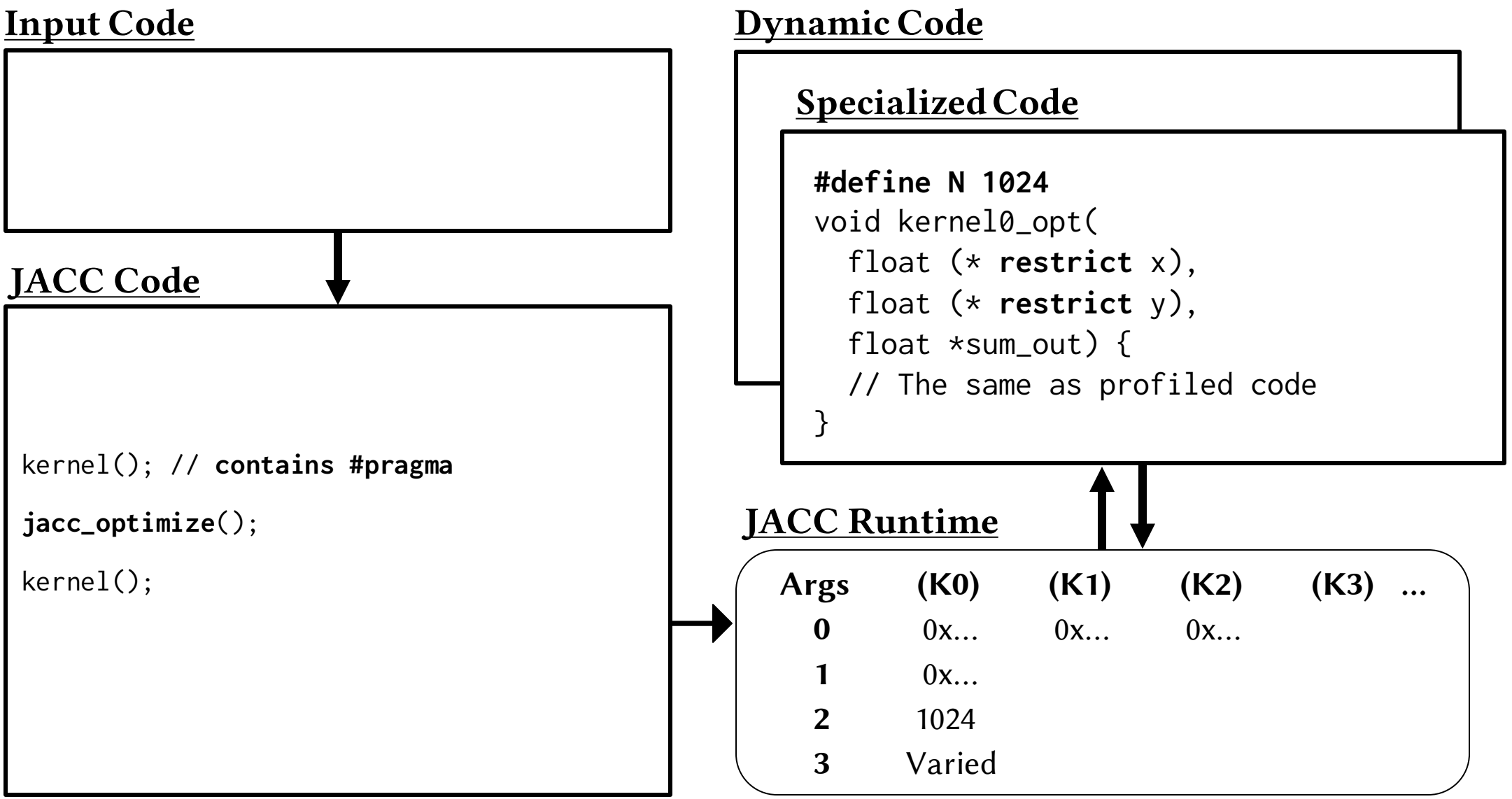
On-the-Fly Kernel Specialization



On-the-Fly Kernel Specialization



On-the-Fly Kernel Specialization



¿Multi-GPU Utilization?

- Previous work have persistently focused on loop splitting over plural GPUs or required manual efforts
- Mere loop splitting causes *unsolvable dependencies* among devices in real applications
- Unified memory can be employed among GPUs, but memory thrashing is inevitable
- **Key ideas to automate multi-GPU use**
  - Increased P2P-Bandwidth/Mem-Bandwidth
  - Less overheads of computation compared to memory accesses

Predicate-Based Filtering

- We duplicate the program structure on all the GPUs
- Our technique **predicate-based filtering** limits memory accesses depending on data regions to which the GPU writes
- GPU-to-GPU communication after each kernel execution

```
a[i]*=2;
```

```
(a_lb <= i && a_ub >= i) ? a[i]*=2 : a[i];
```

lb: lower-bound, ub: upper-bound

```
1  a[i]=x; b[i]=a[i]; x=c[j]
2  a[k]=x; b[k]=a[k];
```

```
1  /* a[i]=x */
2  ((a_lb<=i && a_ub>=i)||
3   (b_lb<=i && b_ub>=i)) ? a[i]=x:a[i];
4  /* b[i]=a[i] */
5  ((b_lb<=i && b_ub>=i)) ? b[i]=a[i]:b[i];
6  /* x=c[j] */
7  x=((a_lb<=k && a_ub>=k)||
8     (b_lb<=k && b_ub>=k)) ? c[j]:0;
9  /* a[k]=x */
10 ((a_lb<=k && a_ub>=k)||
11  (b_lb<=k && b_ub>=k)) ? a[k]=x:a[k];
12 /* b[k]=a[k] */
13 ((b_lb<=k && b_ub>=k)) ? b[k]=a[k]:b[k];
```

Predicate for Multi-Dimensional Arrays

```
1  #pragma acc parallel loop gang
2  for (i = 1; i <= gp02; i++) {
3   #pragma acc loop worker vector
4   for (k = 1; k <= gp22; k++) {
5    for (m = 0; m < 5; m++)
6     for (n = 0; n < 5; n++) {
7      lhsY[n][m][BB][jsize][i][k] =
8        lhsY[n][m][BB][jsize][i][k]
9        - lhsY[n][0][AA][jsize][i][k]
10       * lhsY[0][m][CC][jsize-1][i][k]
11    /* - lhsY[n][1..3][AA][jsize][i][k]
12       * lhsY[1..3][m][CC][jsize-1][i][k] */
13       - lhsY[n][4][AA][jsize][i][k]
14       * lhsY[4][m][CC][jsize-1][i][k];
15     }}}
```

Kernel code from NPB-BT
(Two inner loops are unrolled in actual code)

- Linear splits cause all-to-all dependencies among statements

  → all statements are *duplicated on all the GPUs*

- **We divide a specific dimension**
  - The dimension should contain the most parallel iterators (such as i and k)
  - The leftmost dimension is preferred in the C language and the rightmost dimension in Fortran

Translation for Predicate (NPB-CG)

Original

```
1  #pragma acc parallel/
2    num_gangs(end) num_workers(4)
3    vector_length(32)
4  {
5    #pragma acc loop gang
6    for (j = 0; j < end; j++) {
7      tmp1 = rowstr[j];
8      tmp2 = rowstr[j+1];
9      sum = 0.0;
10     #pragma acc loop worker/
11           vector reduction(+:sum)
12     for (k = tmp1; k < tmp2; k++) {
13       tmp3 = colidx[k];
14       sum = sum + a[k]*p[tmp3];
15     }
16     q[j] = sum;
17   }
18 }
```

Basic Idea:
- Split Array-Writes Evenly
- Filter Access to Dependencies
- All-to-All Communication

Predicated

```
1  #pragma acc parallel num_gangs (end)/
2        num_workers(4) vector_length(32)
3  #pragma acc loop gang
4  for(j = 0; j < end; j++) {
5    tmp1 = ((r_lb <= j) && (r_ub >= j)) ?
6                          rowstr[j]     : 0;
7    tmp2 = ((r_lb <= j) && (r_ub >= j)) ?
8                          rowstr[j + 1] : 0;
9    sum = 0.0;
10   #pragma acc loop worker vector/
11                       reduction(+:sum)
12   for (k = tmp1; k < tmp2; k++) {
13     tmp3 = ((r_lb <= j) && (r_ub >= j)) ?
14                              colidx[k] : 0;
15     sum = sum +
16      (((r_lb <= j) && (r_ub >= j)) ?
17                                 a[k] : 0) *
18      (((r_lb <= j) && (r_ub >= j)) ?
19                                 p[tmp3]:0);
20   }
21   ((r_lb <= j) && (r_ub >= j)) ?
22                        (r[j] = sum) : r[j];
23 }
```

Predicate with Adaptive Algorithm

- **Expected performance degradation by:**
  - Large GPU-to-GPU communication latency
  - Failure of parallelization due to intra-kernel dependencies or less memory accesses

- **Enable multi-GPU execution for each kernel after satisfying the condition below five times:**

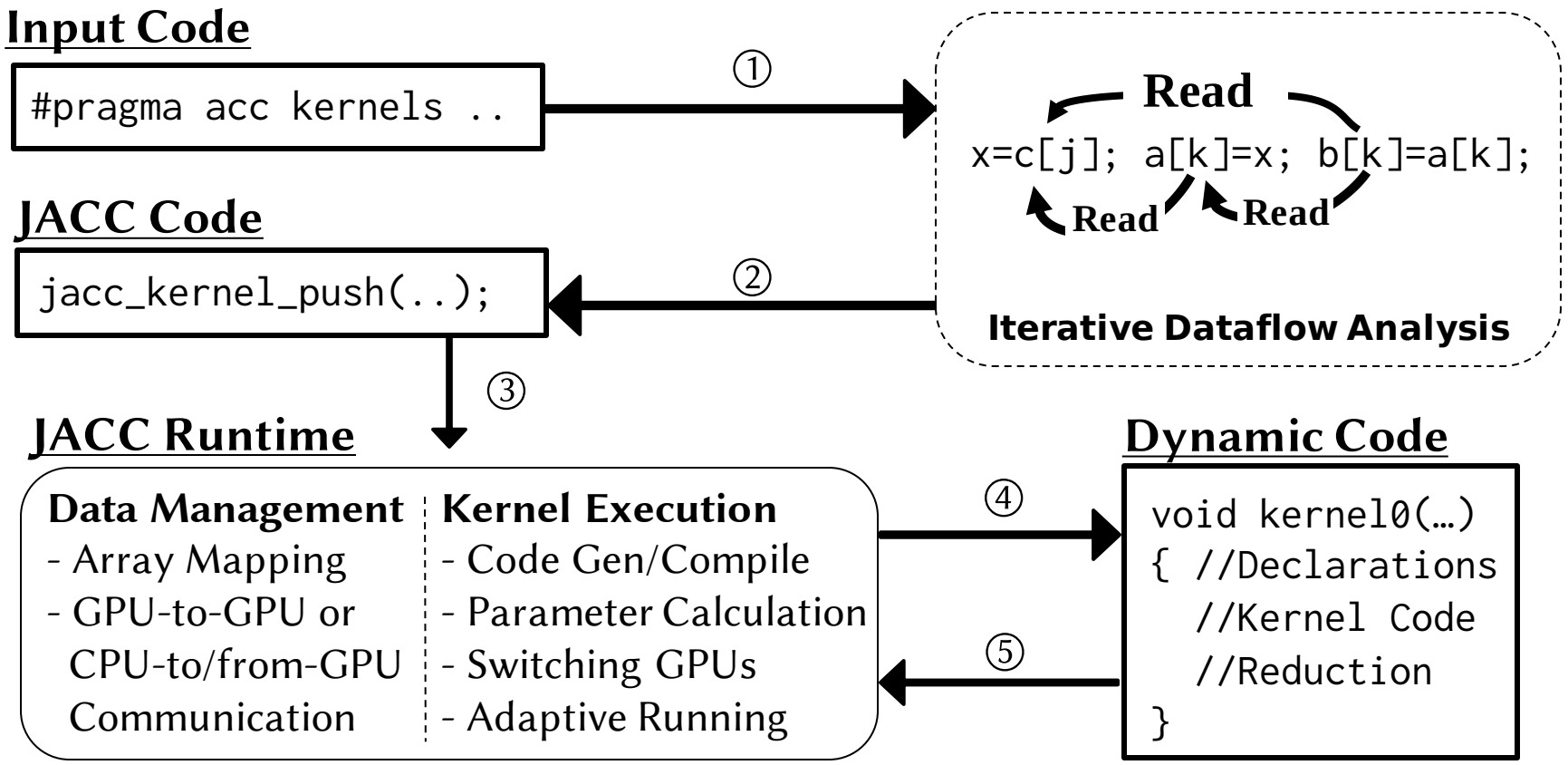
$$time_{\text{Kernel}} > time_{\text{Kernel}}/n_{\text{GPUs}} + \text{WriteSize}/peak_{\text{P2P}}$$

- **Disable multi-GPU execution after satisfying one of conditions five times and when (left−right) > 0 on average**

$$time_{\text{Kernel}} + time_{\text{Comm}} > time_{\text{Kernel}} \times n_{\text{GPUs}}$$

$$time_{\text{Kernel}} + time_{\text{Comm}} > eff_{\text{dup}} \times \text{WriteSize}$$

Implementation of Predicate



Fortran Example

Input Code CloverLeaf calc_dt_kernel.f90

```
!$ACC KERNELS
! …
!$ACC LOOP INDEPENDENT REDUCTION(min:dt_min_val)&
!$ACC GANG(128)
  DO k=y_min,y_max
!$ACC LOOP INDEPENDENT REDUCTION(min:dt_min_val)
    DO j=x_min,x_max
      IF(dt_min(j,k).LT.dt_min_val) &
&                dt_min_val=dt_min(j,k)
    ENDDO
  ENDDO
!$ACC END KERNELS
```

JACC Code

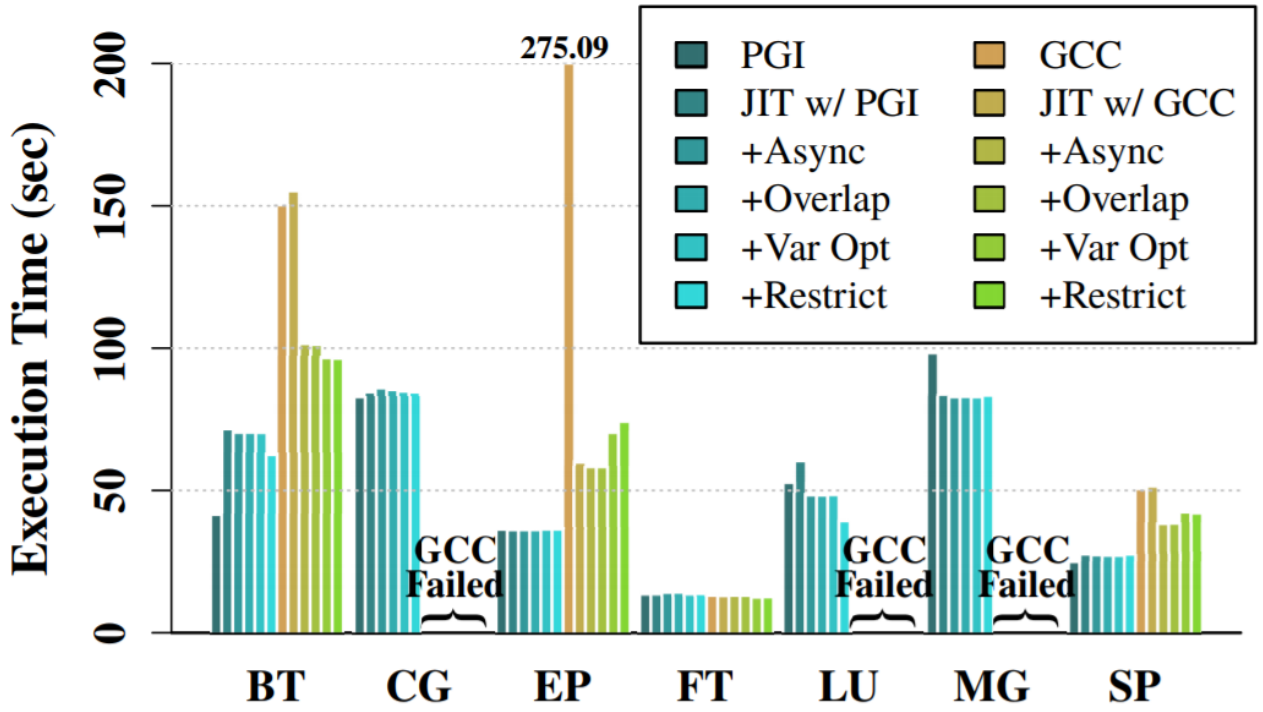
```
BLOCK
  INTEGER(KIND=8) :: arg = 0
  CALL jacc_arg_build(&
    &"real\0", "dt_min\0“,            &! Type, Symbol
    & c_loc(dt_min), sizeof(dt_min),  &! Address, Size
    & …, &! Analysis Info (e.g. WRITE, PRESENT, REDUCTED)
    & lbound(dt_min), ubound(dt_min), &! Boundary
    & arg)
  CALL jack_arg_build(…)
  …
  CALL jacc_kernel_push(&
  && ! Kernel Code
  "  !$ACC PARALLEL PRESENT(dt_min) NUM_GANGS(128)\n &
  & # 139 "calc_dt_kernel.f90"\n &
  & … !$ACC END PARALLEL\0“, arg)
END BLOCK
```

Methodology

- The JACC translator is implemented as a XcodeML converter (C/Fortran ⟶ XML ⟶ C/Fortran). Compiler: PGI/GCC
- The evaluation is performed on NVIDIA DGX-1 w/ Tesla V100 SXM2 (16GB Memory) x 8
  - We use tightly-coupled four GPUs where each link offers a unidirectional bandwidth of 25GB/s
- Benchmarks: Manually-tuned NAS Parallel Benchmarks (NPB) in C & CloverLeaf/CCS-QCD/Himeno in Fortran

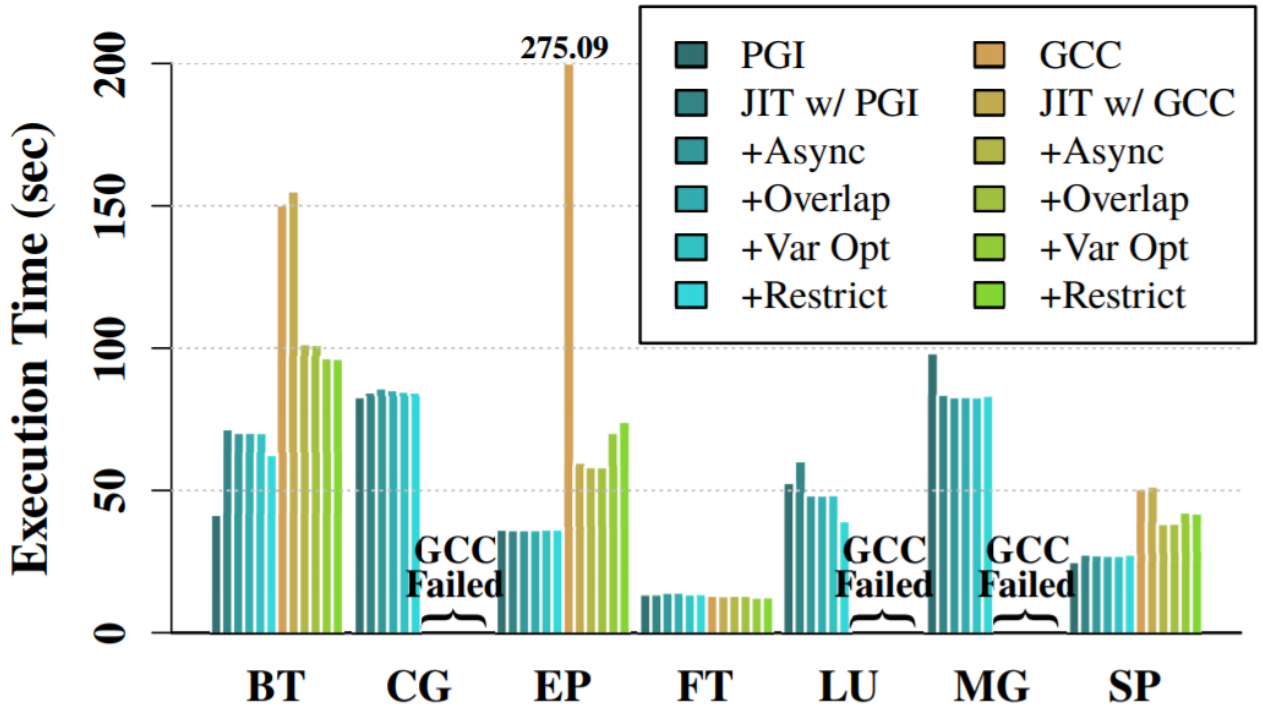
Name	Description	Dependency	Problem Size	Memory	Num Kernels
BT	CFD with Block Tri-Diagonal Solver	Halo (3D)	Class C: 162x162x162 (FP64), iter=200	4.915 GB	46
CG	Minimum-Eigenvalue Calculation	Irregular	Class C: size=150000 (FP64), iter=3750	0.747 GB	16
EP	Random-Number Generation in Parallel	None	Class D: size=137438953472 (FP64)	2.305 GB	4
FT	Discrete 3D Fast Fourier Transform	All-to-All	Class C: 512x512x512 (FP64), iter=20	9.317 GB	12
LU	CFD with Lower-Upper Gauss-Seidel Solver	Halo (3D)	Class C: 162x162x162 (FP64), iter=250	1.471 GB	59
MG	Multigrid Discrete Poisson Equation	Long & Short	Class C: 512x512x512 (FP64), iter=1000	6.114 GB	16
SP	CFD with Scalar Penta-Diagonal Solver	Halo (3D)	Class C: 162x162x162 (FP64), iter=400	1.700 GB	65
CloverLeaf	2D Euler Equations Solver	Halo (2D)	3840x1920 (FP64), step=1800	1.749 GB	114
CCS-QCD	Lattice QCD Simulation	Halo (3D)	32x32x32x128 (FP64), iter=1000 for BiCGStab	15.255 GB	27
Himeno	19-point Jacobian Stencil Computation	Halo (3D)	Size XL:1024x512x512 (FP32), iter=1000	14.409 GB	2

Result: Async/Kernel-Opt



- Performance changes of **Original** → **JACC** due to less static information
- **JACC** → **+Async** Performance improve 3.43% with PGI and 22.08% with GCC on average
  - **+Async** → **+Overlap** No improvement because of little inter-kernel parallelism
- With **+Restrict**, better performance up to 23.39% in BT/LU with PGI and 5.59% less in EP with GCC

Result: Async/Kernel-Opt

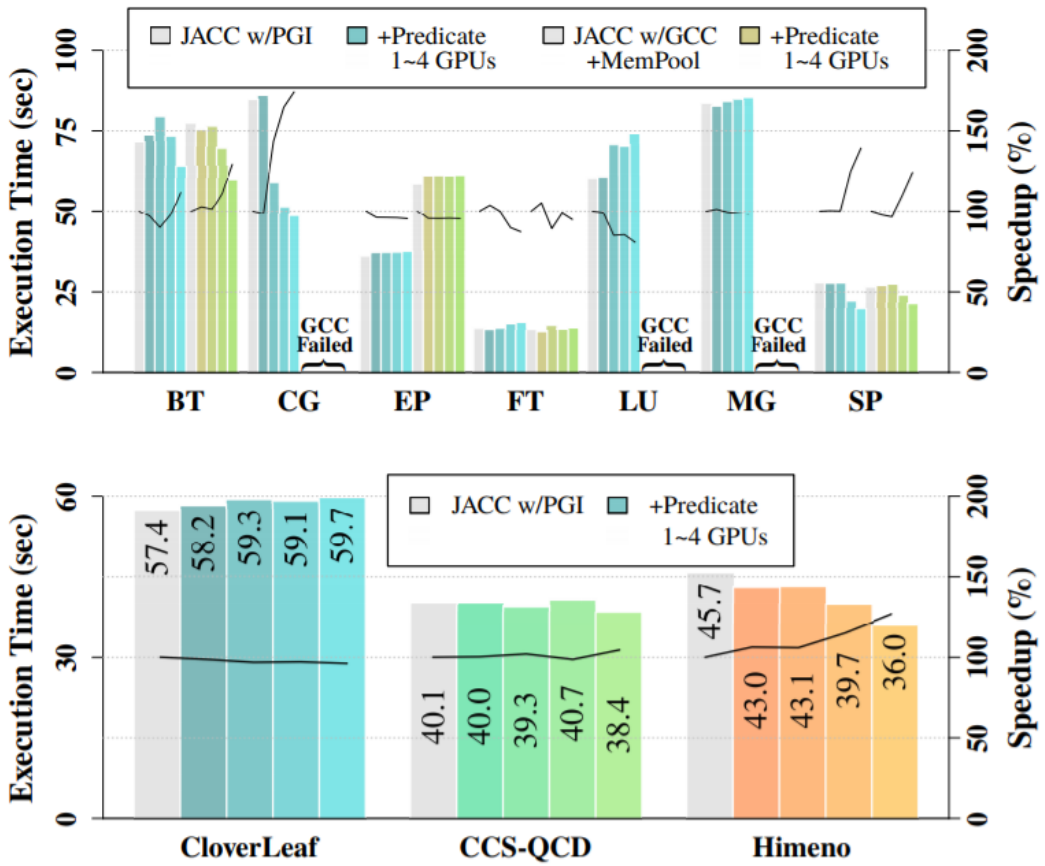


- With +**Var Opt**, no performance change in most cases and worsen efficiencies in EP/SP with GCC
  - Suffered from low utilization by ineffective threads which are created due to reduced register use
  - On the other hand, performance improvements of +**Restrict** are achieved by parallelized memory accesses which require additional registers

Result: GCC Custom Allocation

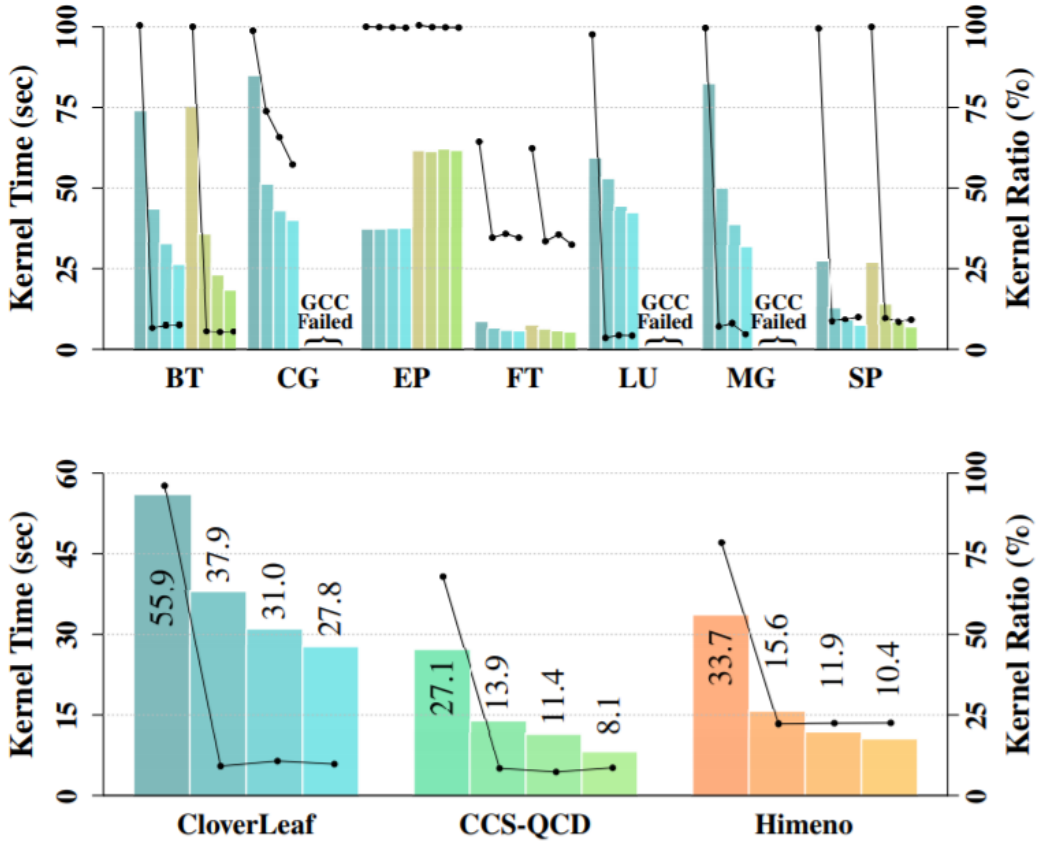
- The performance difference between PGI and GCC is primarily caused by the latency of memory allocation; PGI owns memory pools for device memory, while GCC does not
- For our multi-GPU experiments to explicitly show the performance improvements by kernel parallelization, we **integrate two pools for device memory** into the GCC library libgomp
  - One for user-invoked allocation, another for the GCC's internal allocation which tends to be smaller
  - With the memory-pool integration, GCC's efficiency becomes competitive to PGI while having -7.83% ~ 5.05% better throughputs
  - Except the case of EP, where the kernel execution poses a 38.31% overhead due to GCC's device-code

Result: Multi-GPU Utilization in Total



- Five of 10 applications improved by 4.05% ~ 43.43% when enabling four GPUs compared one GPU use
- When only the kernel execution and GPU-to-GPU transfers are concerned, six benchmarks improved by 23.9% on average
  - Other factors necessary for multi-GPU such as memory allocation and synchronization causes some overheads

Result: Multi-GPU Utilization in Kernel



- Using the memory-intensive benchmarks BT/SP, PGI achieves 2.83x and 3.59x improvements on four GPUs and GCC does 4.13x and 3.85x, respectively
- For LU, the shorter running time of each kernel execution than 1ms limits acceleration to 1.40x, involving overheads for duplicating program
- EP does not improve due to is compute-bound nature
- The 20% execution of CloverLeaf is distributed over multi-GPU but those kernels are not well enhanced

Result: Multi-GPU Utilization in Detail

Name	Num Kernels	Kernel Dup [ms]	Num Adapted	Comm + Kernel [ms]	Kernel Total (ave) [ms]	Kernel Adapted (ave) [ms]	Comm (ave) [ms]	Average WriteSize	GPU-to-GPU Bandwidth
BT	46	88,715	3	**63,856**	46,639 (0.44)	14,940 (24.75)	17,217 (28.52)	684.10 MB	23.99 GB/s
CG	16	75,178	7	**44,137**	38,720 (0.08)	34,365 (0.12)	5,417 (0.02)	0.10 MB	5.40 GB/s
EP	4	37,485	3	37,787	37,710 (24.55)	37,710 (24.55)	77 (0.05)	0.03 MB	0.54 GB/s
FT	12	8,472	4	8,757	6,096 (47.29)	4,983 (62.37)	2,661 (33.31)	806.92 MB	24.23 GB/s
LU	59	76,325	7	**71,614**	64,342 (0.09)	3,886 (0.03)	7,272 (0.06)	0.39 MB	6.28 GB/s
MG	16	83,586	3	83,654	83,654 (0.47)	5,604 (0.35)	0 (0.00)	0.00 MB	0.00 GB/s
SP	65	27,809	3	**19,609**	16,648 (0.64)	1,969 (1.64)	2,961 (2.47)	58.24 MB	23.60 GB/s
CloverLeaf	114	55,017	3	55,272	54,079 (0.22)	10,980 (4.27)	1,193 (0.46)	5.46 MB	11.76 GB/s
CCS-QCD	27	26,517	11	**24,641**	13,031 (0.97)	5,811 (1.91)	11,610 (3.82)	91.67 MB	24.02 GB/s
Himeno	2	33,726	1	**23,149**	11,894 (5.93)	8,318 (8.30)	11,255 (11.23)	271.47 MB	24.18 GB/s

- Better scaling can be obtained with longer kernel execution even with larger transfers
  - BT is successfully parallelized with communications of a six-dimensional array decomposed per GPU in 75 segments of 8MB size, having original kernel execution longer than 10ms

Result: Data-Size Scaling with Predicate

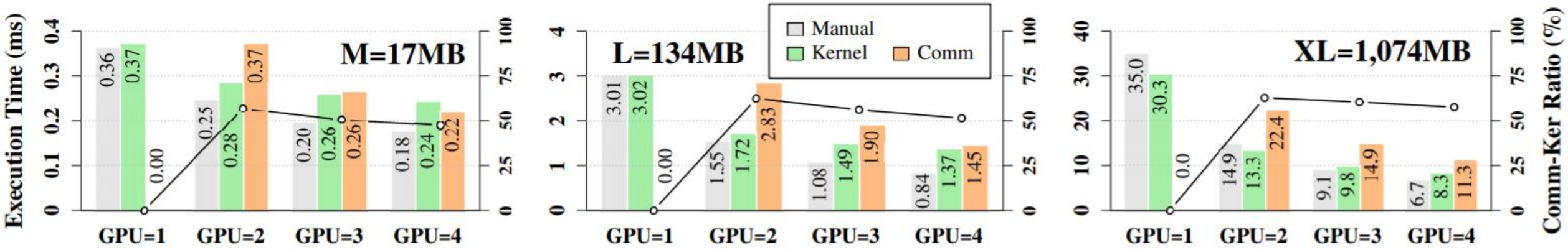


Fig. Performance of Himeno Benchmark (Stencil Code)

- Communication speed with two GPUs → four GPUs, 1.70x with size M, 1.95x with L and 1.99x with XL
- Kernel Performance with single → four GPUs, 1.53x, 2.19x and 3.64x improvements for size M, L and XL
- The optimized communication for the stencil application significantly reduces the latencies.

  We consider those domain-specific approaches for future work

Result: Comparison with Predicate

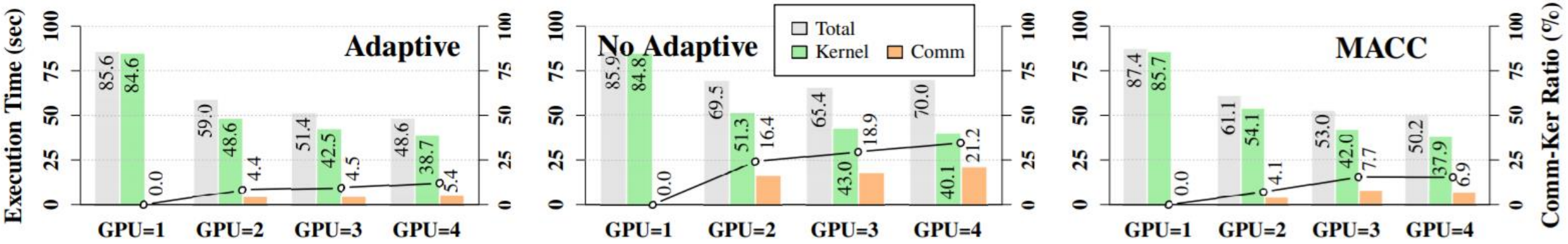


Fig. Performance of NPB-CG

- Previous work MACC can support only two of 10 benchmarks (Himeno/CG)
- MACC automates loop splitting of all the kernels in CG but employs no adaptive algorithm
- Our algorithm has better efficiency up to 3.50% by disabling multi-GPU execution for lower-latency kernels
- Compared to non-adaptive execution, we archive 44.09% better total efficiency

Conclusion

- We presented **JACC**, an OpenACC framework which facilitates runtime extension
  - JACC creates dynamic code from original kernels and compiles it with a specified compiler in order to support on-the-fly code extension automatically
  - Demonstrated for asynchronous execution and kernel specialization
- We proposed **predicate-based filtering**, a novel code-translation technique of multi-GPU utilization, for distributing highly-tuned applications without additional user effort
- Having many kernels parallelized on a multi-GPU environment, we showed the performance improvements of several tested benchmarks where precise data-dependency analysis is always restrained

BACK UP

EPEEC
PRODUCTIVITY AT EXASCALE