An Extension of OpenACC Directives for Out-of-Core Stencil Computation with Temporal Blocking

Nobuhiro Miki   Fumihiko Ino   Kenichi Hagihara

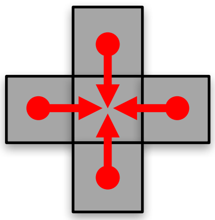
Graduate School of Information Science and Technology

Osaka University

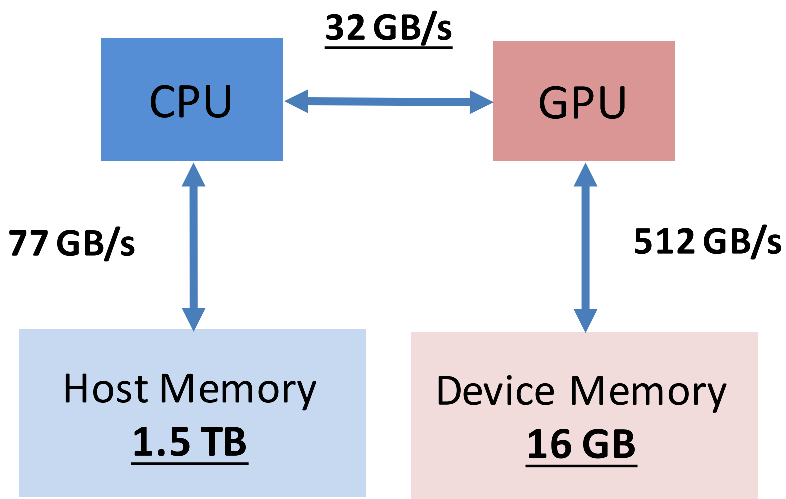
Stencil computation in OpenACC

- Stencil computation
  - A fixed pattern is iteratively applied to every data elements to solve time evolution equations
  - Usually accelerated on a GPU equipped with high memory bandwidth
- OpenACC: the simplest method for developing GPU code
  - Useful to separate accelerator-specific code from CPU code
- OpenACC is not a perfect solution for out-of-core data
  ① Limited problem size due to exhaustion of GPU memory
  ② Time evolving iterations can transfer many data between CPU and GPU

```
for (t=0; t<T; t++) { // time evolution loop
 #pragma acc kernels loop
 for(i=0; i<N; i++)
   #pragma acc loop
   for(j=0; j <N; j++)
      a[i][j] = a[i][j-1] + a[i-1][j]
              + a[i][j+1] + a[i+1][j];
}
```



Stencil code in OpenACC



Memory architecture

Out-of-core code with temporal blocking

- Data decomposition and temporal blocking are useful for tackling these issues
- The performance portability is degraded due to code modification
  - Accelerator-specific code is mixed with the essence of computation

```
allocate buf_p[0], ..., buf_p[num_queue] on host memory;
#pragma acc create (buf_p [0:num_queue] [0:b+2*h*k], ...)

for (n=0; n<T; n+=k)
  for (c=0; c<d; c++) {

   set si as the id of an idle queue; // 0 <= si <num_queue
   copy chunk from p to buf_p[si
   #pragma acc update device (b

   for (i=0; i<k; i++) {
    #pragma acc ke          present (buf_p[si:1][0:b+2*h*k],...) async(si)
    {
     offset =
     xsize = b+2*h*(k-1-i);
     #pragma acc loop independent
     for (x=offset; x<offset+xsize; x++)
      #pragma acc loop independent
      for (y=1; y<y-1; y++)
       #pragma acc loop independent
       for (z=1; z<z-1; z++)
        buf_q[si][x*y*z+y*z+z] += buf_p[si][(x+1)*y*z+y*z+z] + ...;
    }
    buf_p[si] = buf_q[si];
   }
   #pragma acc update host (buf_p [si:1][0:b+2*h*k], ...) async (si)
   copy chunk from buf_p[si] to p;
  }
}
```

Allocate buffers in both host and device

Select an asynchronous queue

Modify loop structures

Modify indexing scheme

Overview

- Goal: to facilitate data decomposition and temporal blocking for GPU-accelerated stencil computation
- Method: directive-based approach
  ① Pipelined accelerator (PACC): an extension of OpenACC directives
  ② Source-to-source translator for PACC -> OpenACC translation

```
#pragma pacc init
#pragma pacc pipeline targetinout(work,a)  size([0:Y][0:X]) halo([1:1][1:1]) async
for(n=0;n<nn;n++){
  #pragma pacc loop dim(2)
  for(x=1;x<X-1;x++)
    #pragma pacc loop dim(1)
    for(y=1;y<Y-1;y++)
      work[x][y] = (a[x-1][y] + ... ) ;
  #pragma pacc loop dim(2)
  for(x=1;x<X-1;x++)
    #pragma pacc loop dim(1)
    for(y=1;y<Y-1;y++)
      a[x][y] = work[x][y];
}
```

<u>Stencil code with PACC</u>

PACC(Pipelined ACCelerator) directives

- PACC extends OpenACC directives with three constructs

```
#pragma pacc init
#pragma pacc pipeline targetinout(work,a) ¥
size([0:Y][0:X]) halo([1:1][1:1]) async
for(n=0;n<nn;n++){
  #pragma pacc loop dim(2)
  for(x=1;x<X-1;x++)
    #pragma pacc loop dim(1)
    for(y=1;y<Y-1;y++)
      work[x][y] = (a[x-1][y] + … ) ;
  #pragma pacc loop dim(2)
  for(x=1;x<X-1;x++)
    #pragma pacc loop dim(1)
    for(y=1;y<Y-1;y++)
      a[x][y] = work[x][y]; }
```

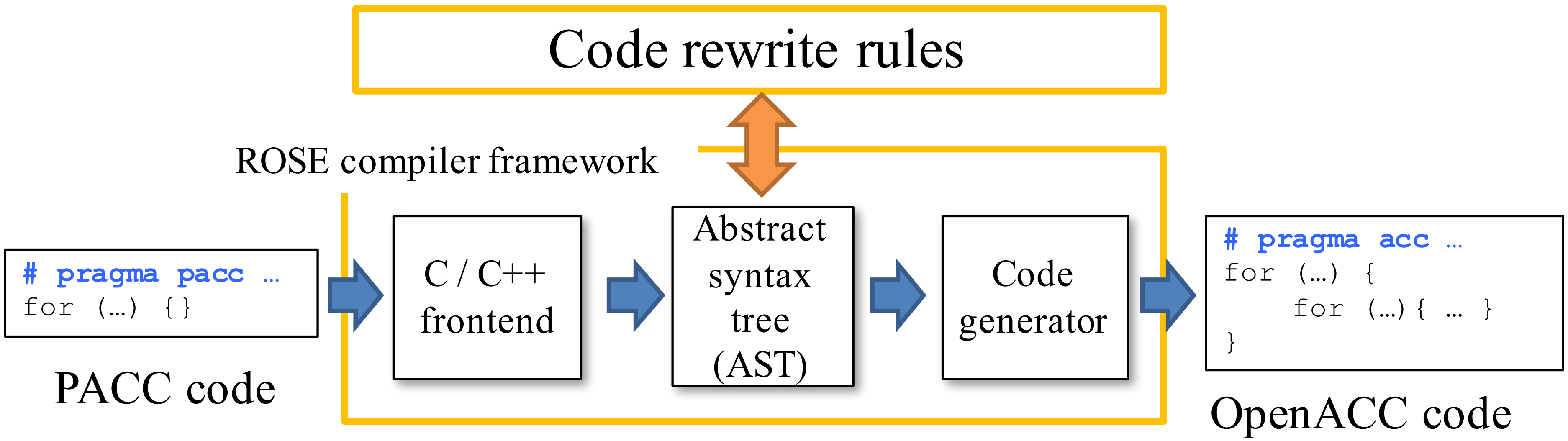
The `init` construct allocates host and device buffers for realizing data decomposition

The `loop` construct indicates which array dimension corresponds to the loop control variable

- The `pipeline` construct specifies the code block to be processed in a pipeline
- This construct can have additional clauses
  - **targetin**
    - names of read-only arrays
  - **targetinout**
    - names of writable arrays
  - **size**
    - array size
  - **halo**
    - halo region size
  - **async**
    - async flag

Overview of PACC translator

1. C/C++ frontend generate an abstract syntax tree (AST) of input code using the ROSE compiler infrastructure [2]
2. The generated AST is then traversed to detect AST nodes that have directive information
3. In the next traversal, these detected nodes are updated according to code rewrite rules, which we implemented for PACC
4. Finally, the transformed AST is given to a code generator, which outputs an out-of-core OpenACC code

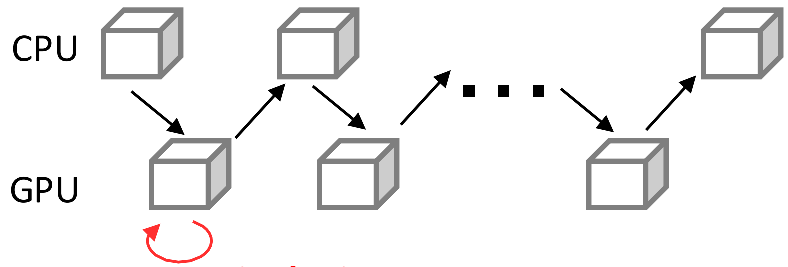


[2] rosecompiler.org. ROSE compiler infrastructure, 2015. http://rosecompiler.org/.

Rewrite Rules for Temporal Blocking

- A cache optimization technique for time evolving computation
  - Computation area is updated k (blocking factor) steps for each data transfer
  - The number of data transfer between CPU and GPU reduces to 1/ k

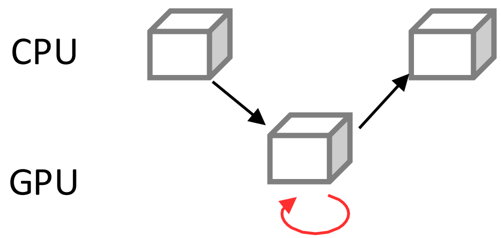
<u>Native implementation</u>



Process a single time step

```
for (n=0; n<T; n++) {
  data transfer from CPU to GPU
  kernel invocation
  data transfer from GPU to CPU
}
```

<u>Apply Temporal Blocking</u>



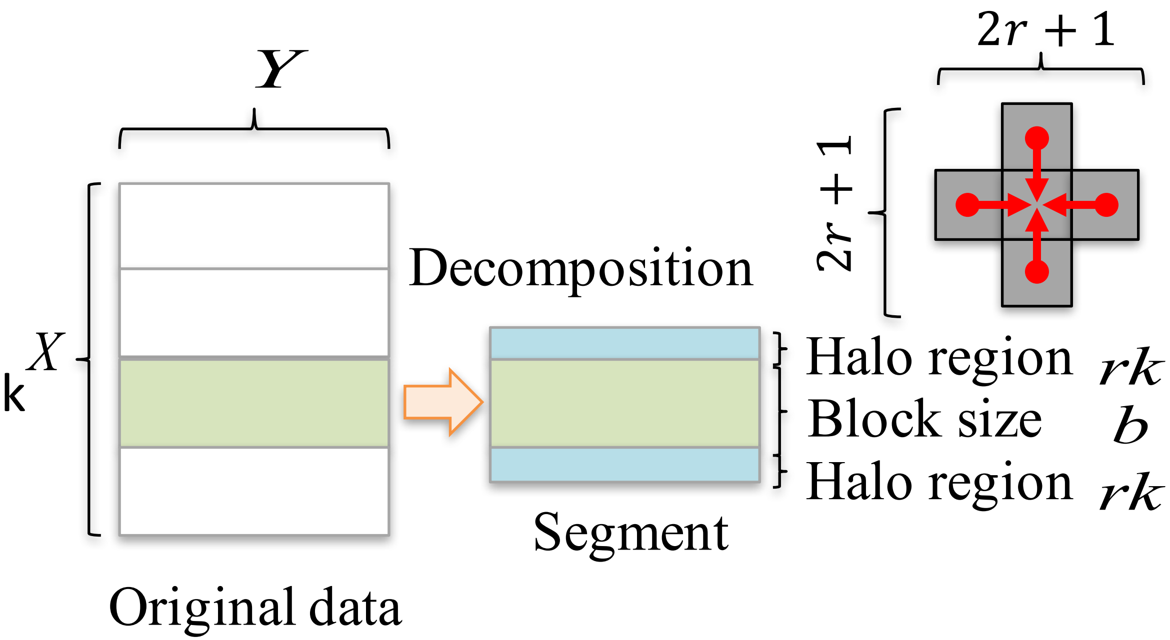
Process *k* time step

```
for (n=0; n<T; n+=k) { // outer loop
  data transfer from CPU to GPU
  for (i=0; i<k; i++) {    // inner loop
    kernel invocation
  }
  data transfer from GPU to CPU
}
```

Data decomposition

- 1-D block scheme
- Given a stencil of $(2r+1)\times(2r+1)$ elements, each block requires halos of size $rk\times Y$ to be processed independently
  - r: the number of neighbor elements in up/down/left/right directions
- Decomposed segments are processed independently
- A software pipeline is used to overlap kernel execution with data transfer
- There are two execution parameters, blocking factor k and block size b

- Block
  - a computation area
- Halo region
  - a boundary area
  - Transferred with Block
- Segment
  - Block + Halo region



Comparison with in-core implementation

- Out-of-core performances were **only 11% - 21% lower** than in-core performance
- If you accept these slowdowns, you can easily process out-of-core data with PACC directives

Experimental machine

- Intel Xeon E5-2680v2 (512 GB)
- NVIDIA Tesla K40 (12 GB)
- Ubuntu 15.3
- CUDA 7.0
- PGI compiler 15.5

Code	Data size		Performance		
	In-core d_1 (GB)	Out-of-core d_2 (GB)	In-core p_1 (GFLOPS)	Out-of-core p_2 (GFLOPS)	**Slowdown** $1 - p_2/p_1$ (%)
Jacobi	4.6	18.4	32.2	28.5	**11**
Himeno	2.3	15.0	47.5	37.5	**21**
CIP	8.2	15.5	83.9	73.4	**13**

Tradeoff relation under CIP method

- The constraint interpolation profile (CIP) method
  - A solver for hyperbolic partial differential equations
  - 9-point 2-D stencil

Tradeoff point

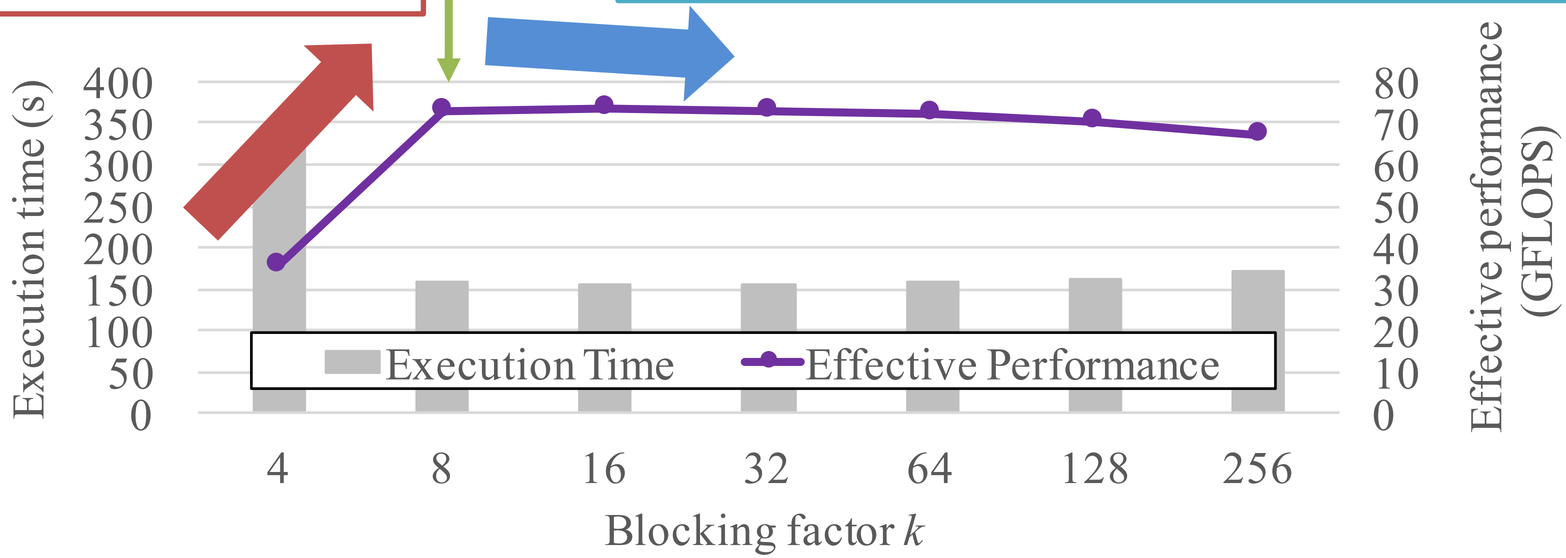
- As we estimated before, the best tradeoff point was found
- The data transfer were fully overlapped with kernel execution

Memory-bound

- Temporal blocking decreased data transfer time

Compute-bound

- Temporal blocking increased kernel execution time slightly due to redundant computation



Conclusion

- PACC: an extension of OpenACC directives capable of accelerating out-of-core stencil computation with temporal blocking on a GPU
  - A translator using AST-based transformation
- Experiments
  - Out-of-core performances were only 11% - 21% lower than in-core performance
  - Tradeoff relation between data transfer time and kernel execution time
- Future work
  - An automated framework for finding best execution parameters (block size and blocking factor)