THE UNIVERSITY OF TOKYO

The University of Electro-Communications UEC

Japan Science and Technology Agency

Satoshi OHSHIMA, Shoichi HIRASAWA, Hiroki HONDA

OMPCUDA : OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler

Outline

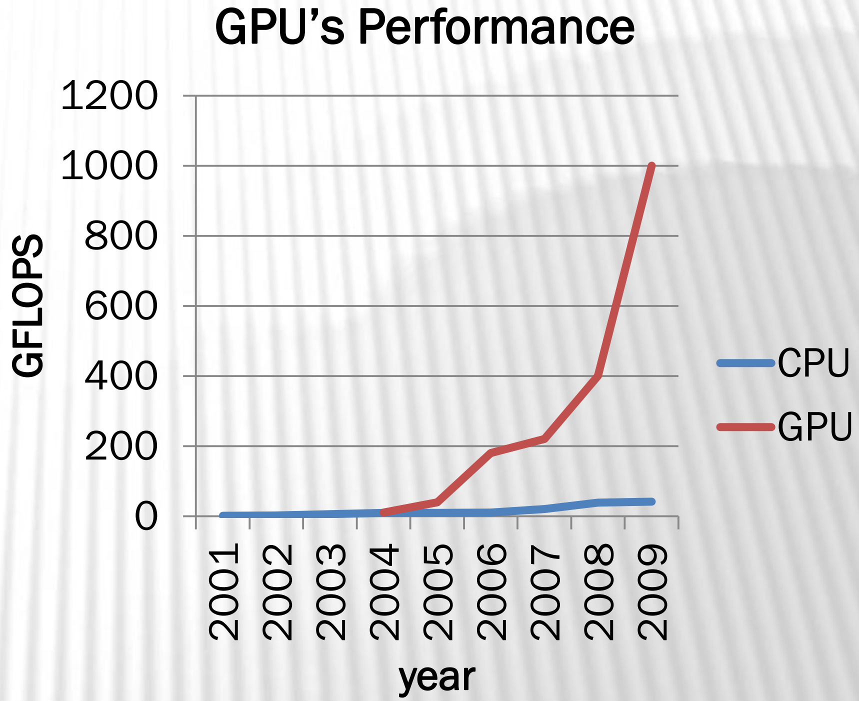
- Motivation
- GPU and CUDA
- Implementation
- Performance evaluation
- Summary

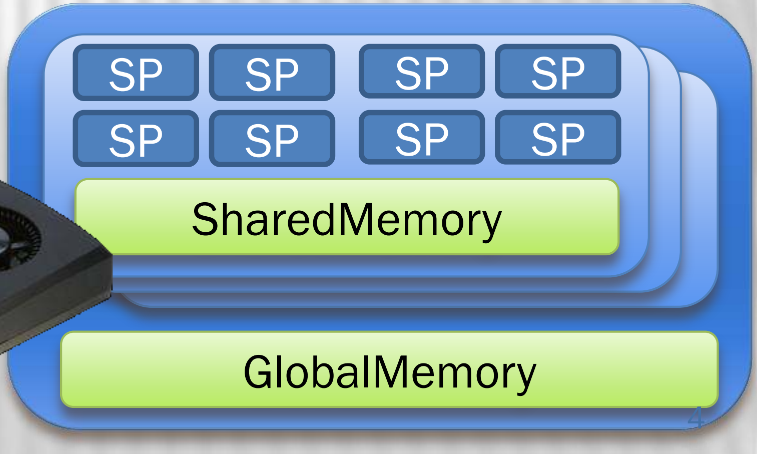
Motivation

- We want to make GPU programming more easily.
  - GPU programming requires specific languages
    - past: Shader (OpenGL+GLSL, DirectX+HLSL)
    - now: CUDA
    - future: CUDA and/or OpenCL ?
  - programmers have to learn new languages and tools
    - time-consuming, heavy
- Can we use exist common parallel programming languages ?
  - As a concrete implementation of our aim, we are now developing an OpenMP framework for CUDA.

GPU and CUDA

- GPU
  - massively parallel hardware
  - very high performance
    - flops/watt, flops/price, flops/volume
  - GPGPU (General-Purpose computing on GPUs)
    - for science, numerical, and multimedia programs
- CUDA
  - architecture and programming environment for NVIDIA GPU
  - provides extended language of C/C++





CUDA : from our point of view…(1/2)

- Fact
  - Many users are using CUDA. The number of users is increasing.
  - Many applications got higher performance than using CPU.
- Question
  - Can all programmers use CUDA ? Is CUDA easy ?
    - Parallel programming is now very important and in demand.
    - But many programmers are already using other languages, such as MPI and OpenMP.

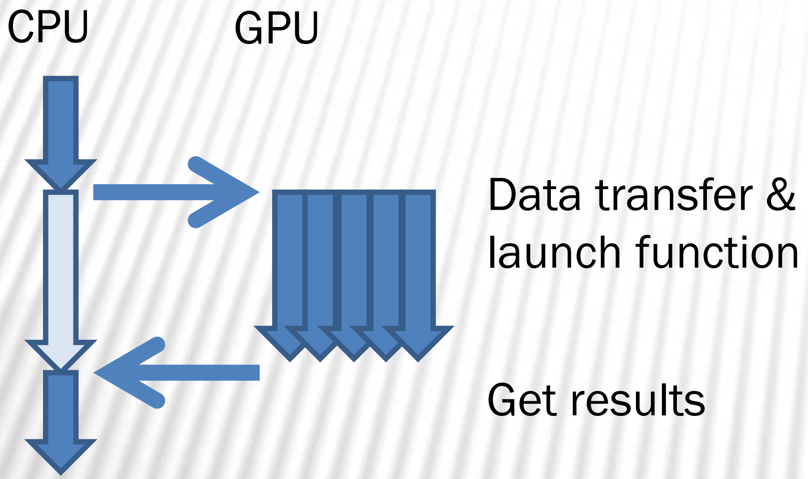
CUDA : from our point of view…(2/2)

- CUDA is not easy, it is difficult and laborious(especially for beginners)
  - the hardware model, memory model, execution model
  - tuning, debugging, ……
  - (I sure acknowledge that CUDA is much easier and clearer than graphics programming based GPGPU.)
  - Can we use exist common parallel programming languages ?

What language (library) matches to CUDA?

(At first we didn't intend to use OpenMP for CUDA, just one of the candidates.)

- Execution model



CPU

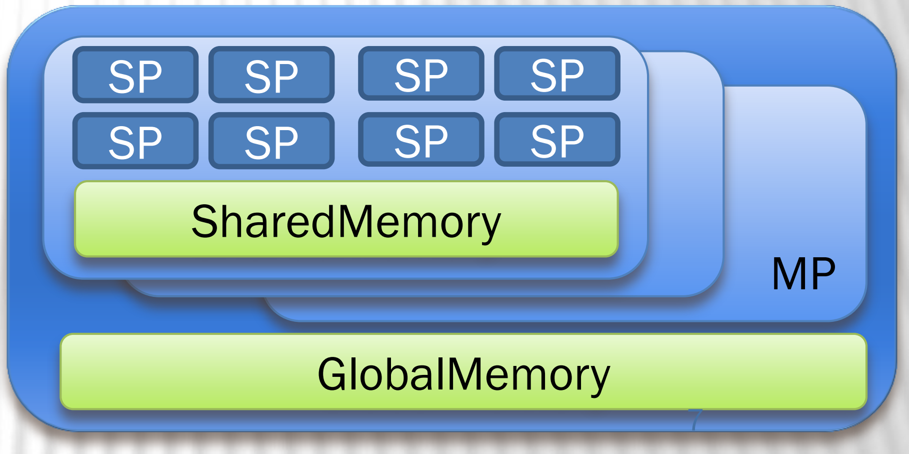
GPU

Data transfer & launch function

Get results

  - execution unit = function
  - many execution instances are launched and killed at once

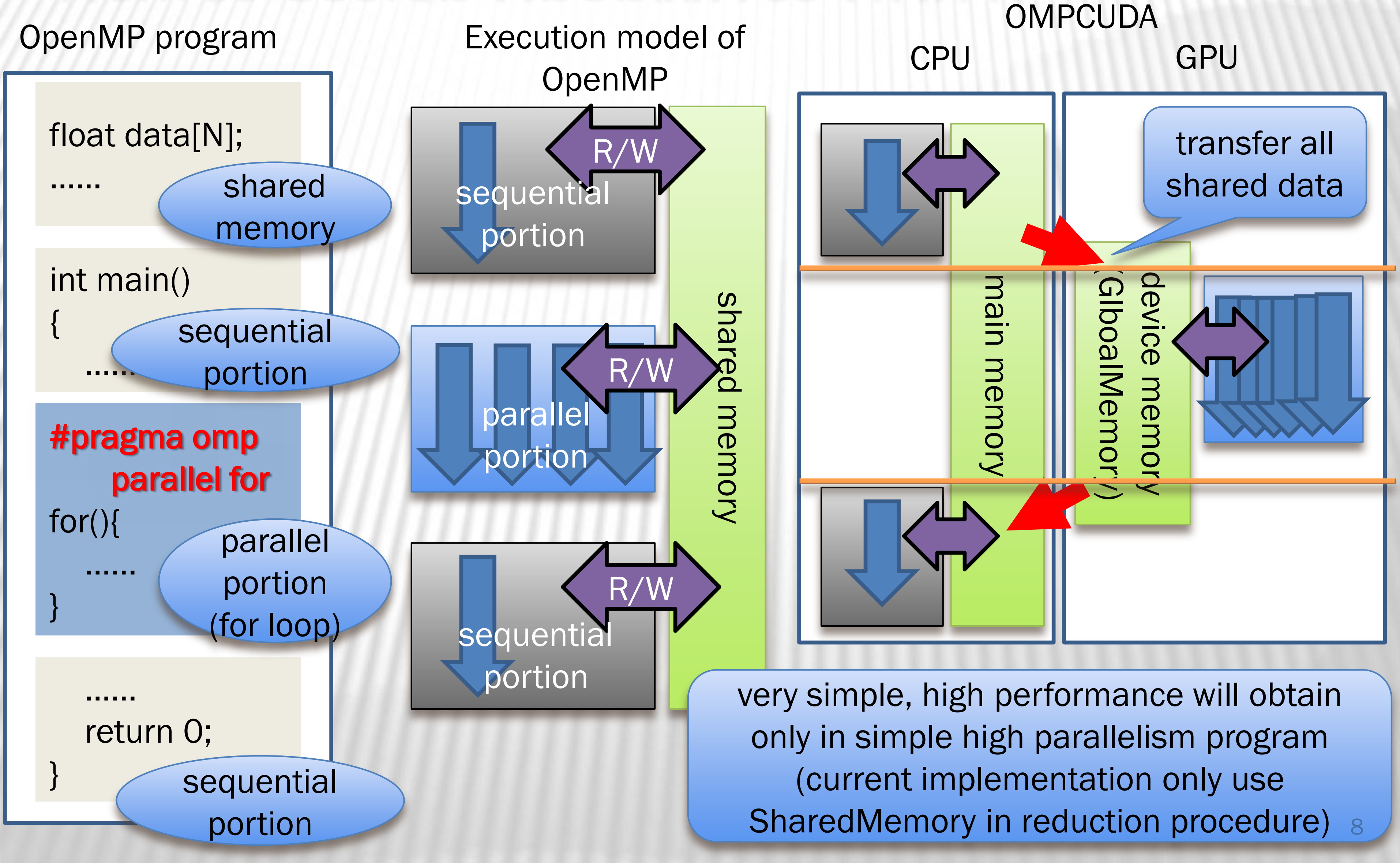
- Hardware model



  - GPU has hierarchical parallelism
  - CPU and GPU have each independent memory

- OpenMP's typical parallel for/DO loop matches CUDA

How to assign OpenMP to GPU ?



OpenMP program

```
float data[N];
……
int main()
{
    ……
#pragma omp
    parallel for
for(){
    ……
}
    ……
    return 0;
}
```

transfer all shared data

very simple, high performance will obtain only in simple high parallelism program (current implementation only use SharedMemory in reduction procedure)

How to make the system ?

- Scratchbuild ?
  - Scratchbuild takes a long time.
  - reinvention of the wheel
  - It is not necessary to implement the OpenMP processor by our own hand.
- Extend and re-create some existing environments
  - We can reduce the time and labor of implementing the OpenMP processor.
  - There are some OpenMP compilers, which compiler can I use it ?

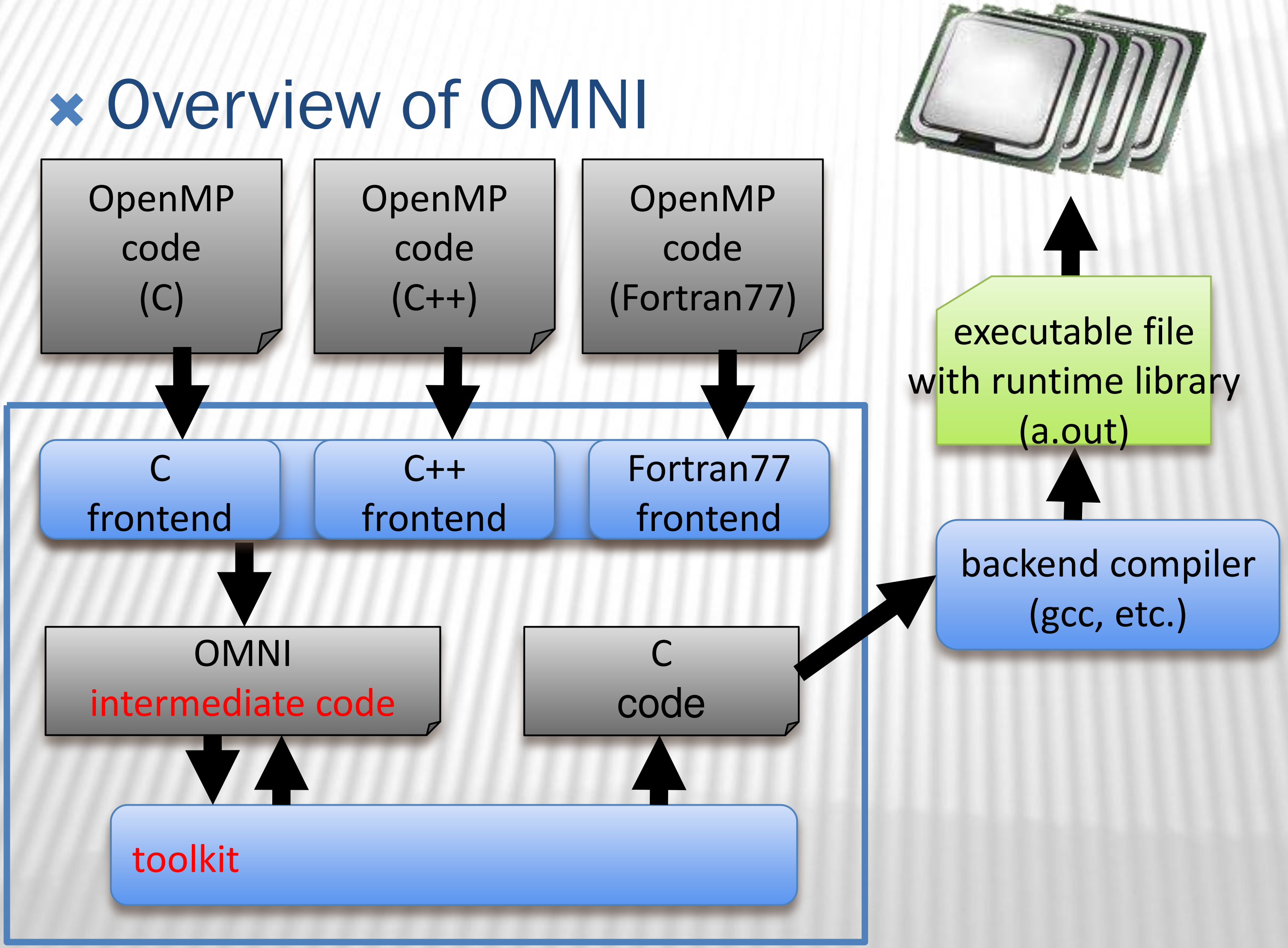
OMNI OpenMP Compiler (OMNI)

- OpenMP compiler developed in Tsukuba
  - published over 10 years ago, and contributed a great deal to the popularization of OpenMP
  - does not support latest OpenMP specifications, but it has some useful features

M.Sato, S.Satoh, K.Kusano, Y.Tanaka: Design of OpenMP Compiler for an SMP Cluster. In: EWOMP '99. (1999) 32–39

OMNI and OMPCUDA

- Overview of OMNI



OpenMP code (C)

OpenMP code (C++)

OpenMP code (Fortran77)

C frontend

C++ frontend

Fortran77 frontend

OMNI intermediate code

C code

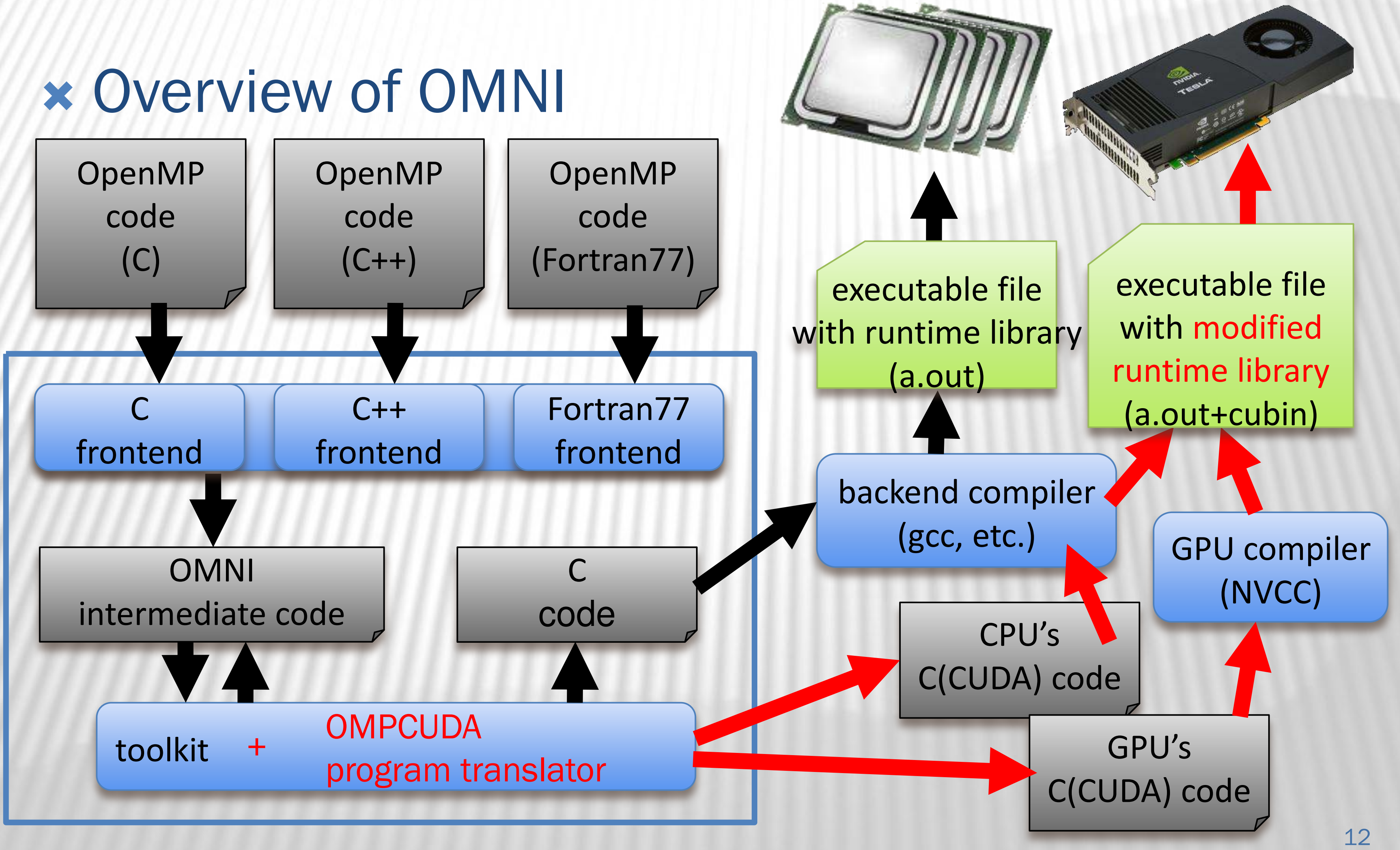
toolkit

backend compiler (gcc, etc.)

executable file with runtime library (a.out)

OMNI and OMPCUDA

- Overview of OMNI



OpenMP code (C)

OpenMP code (C++)

OpenMP code (Fortran77)

C frontend

C++ frontend

Fortran77 frontend

OMNI intermediate code

C code

toolkit + OMPCUDA program translator

backend compiler (gcc, etc.)

executable file with runtime library (a.out)

CPU's C(CUDA) code

GPU's C(CUDA) code

GPU compiler (NVCC)

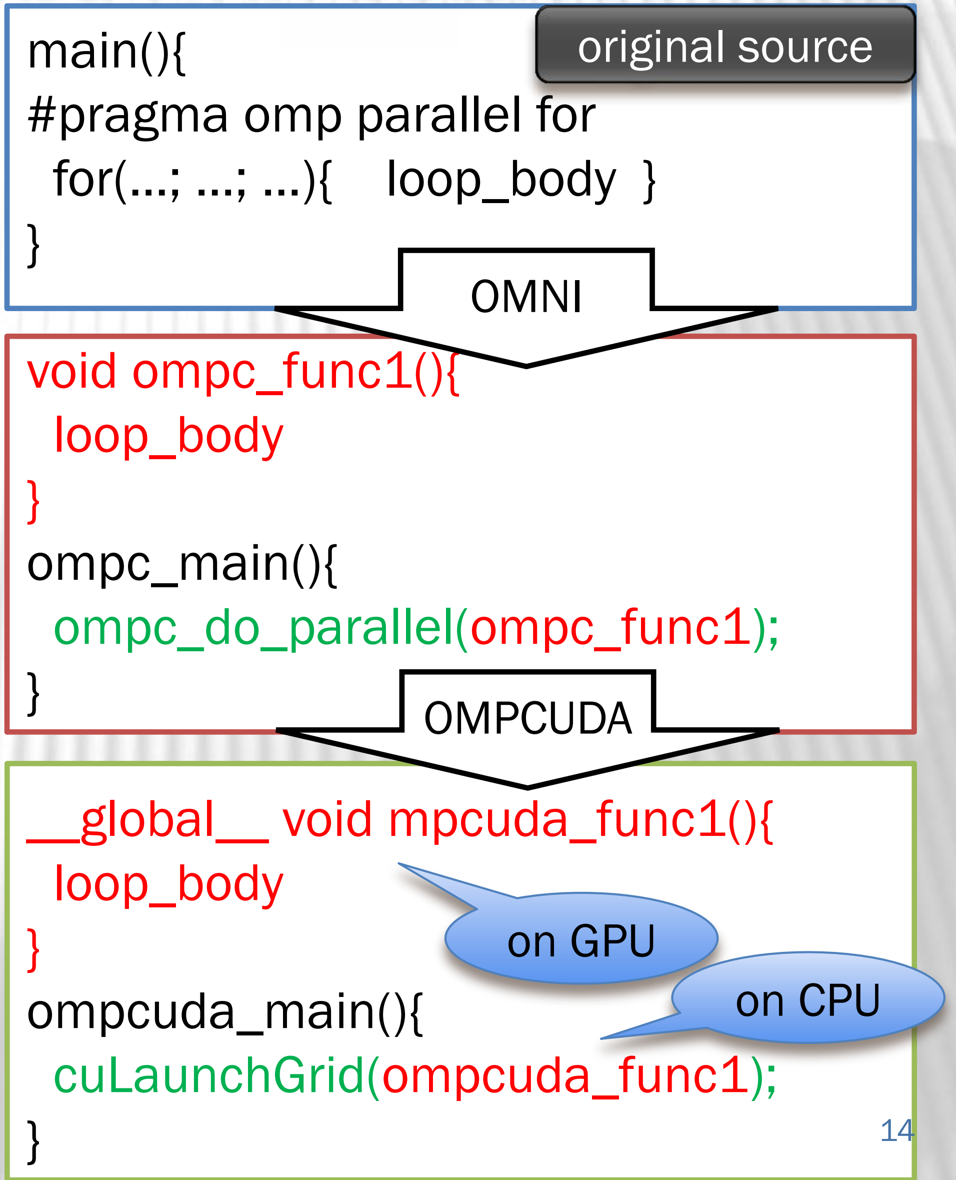
executable file with modified runtime library (a.out+cubin)

Program translator

- important jobs
  1. divide CPU portions and GPU portions
  2. find and transfer shared variables

Divide CPU portions and GPU portions

- in intermediate code of OMNI, GPU portions are rewritten to independent functions and thread launch functions
- OMPCUDA can find GPU portion easily by searching OMNI's thread launch functions



original source

```
main(){
#pragma omp parallel for
  for(...; ...; ...){    loop_body  }
}
```

```
void ompc_func1(){
  loop_body
}
ompc_main(){
  ompc_do_parallel(ompc_func1);
}
```

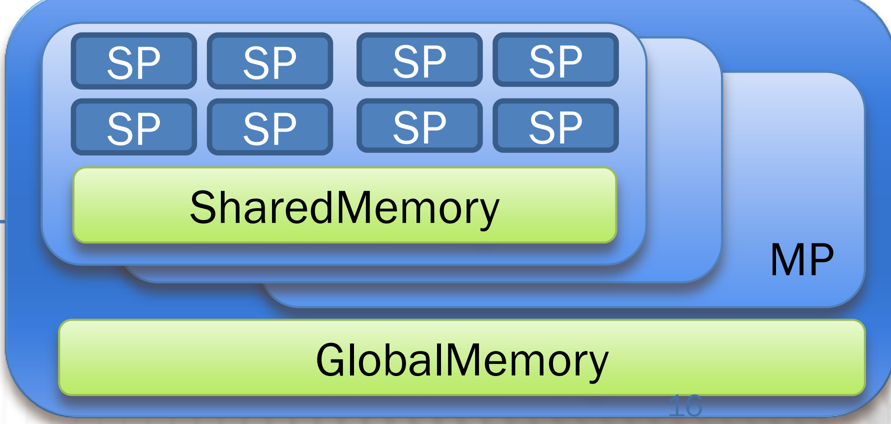
```
__global__ void mpcuda_func1(){
  loop_body
}
ompcuda_main(){
  cuLaunchGrid(ompcuda_func1);
}
```

Find and transfer shared variables

- make steady efforts
  - global variables
    - OpenMP on CPU doesn’t need to transfer, OMPCUDA has to analyze
    - trace intermediate code and check variables
  - local variables
    - OpenMP on CPU need to transfer, local variables have been checked by OMNI

- problems
  - dynamic variables(array and struct), pointers
    - difficult to know size
    - common problem with CPU’s OpenMP, but CPU can execute because of shared memory
    - now OMPCUDA cannot translate and execute complex programs

Runtime library



1. thread management
   - assign OpenMP threads to GPU cores
   - OMNI supports static, dynamic, and guided scheduling
   - OMPCUDA now supports only simple static chunk scheduling (next slide)
2. reduction
   - using SharedMemory (using well-known algorithm)
3. barrier (!)
   - OMNI runtime library handles barrier
   - difficult for OMPCUDA (not implemented yet)
     - CUDA can’t synchronize across the all processors

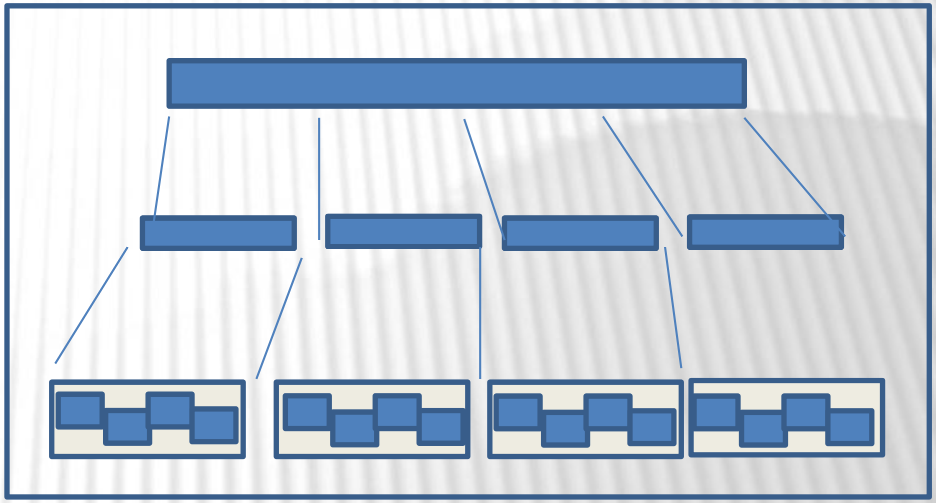
Thread management (Assignment)

original for loop

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

OMNI's default

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31



Simple block division
(Also cyclic division is possible)

OMPCUDA

Blk 0		Blk 1		Blk 2		Blk 3	
Th 0	0 1 2 3	Th 0	16 17 18 19	Th 0	32 33 34 35	Th 0	48 49 50 51
Th 1	4 5 6 7	Th 1	20 21 22 23	Th 1	36 37 38 39	Th 1	52 53 54 55
Th 2	8 9 10 11	Th 2	24 25 26 27	Th 2	40 41 42 43	Th 2	56 57 58 59
Th 3	12 13 14 15	Th 3	28 29 30 31	Th 3	44 45 46 47	Th 3	60 61 62 63

Performance evaluation

- Evaluation environment
  - CPU: Intel XeonE5345 (4core, 2.33GHz)
  - GPU: GeForce GTX 280 (240SP, 1.296GHz)
  - etc.: CUDA Toolkit 2.0, Omni OpenMP Compiler 1.6, CentOS 5.0

- Test programs:
  1. matrix product, single C source code
  2. pi calculation, single C source code (omit in this presentation)
  3. swim(SPEC OMP2001), single F77 source code

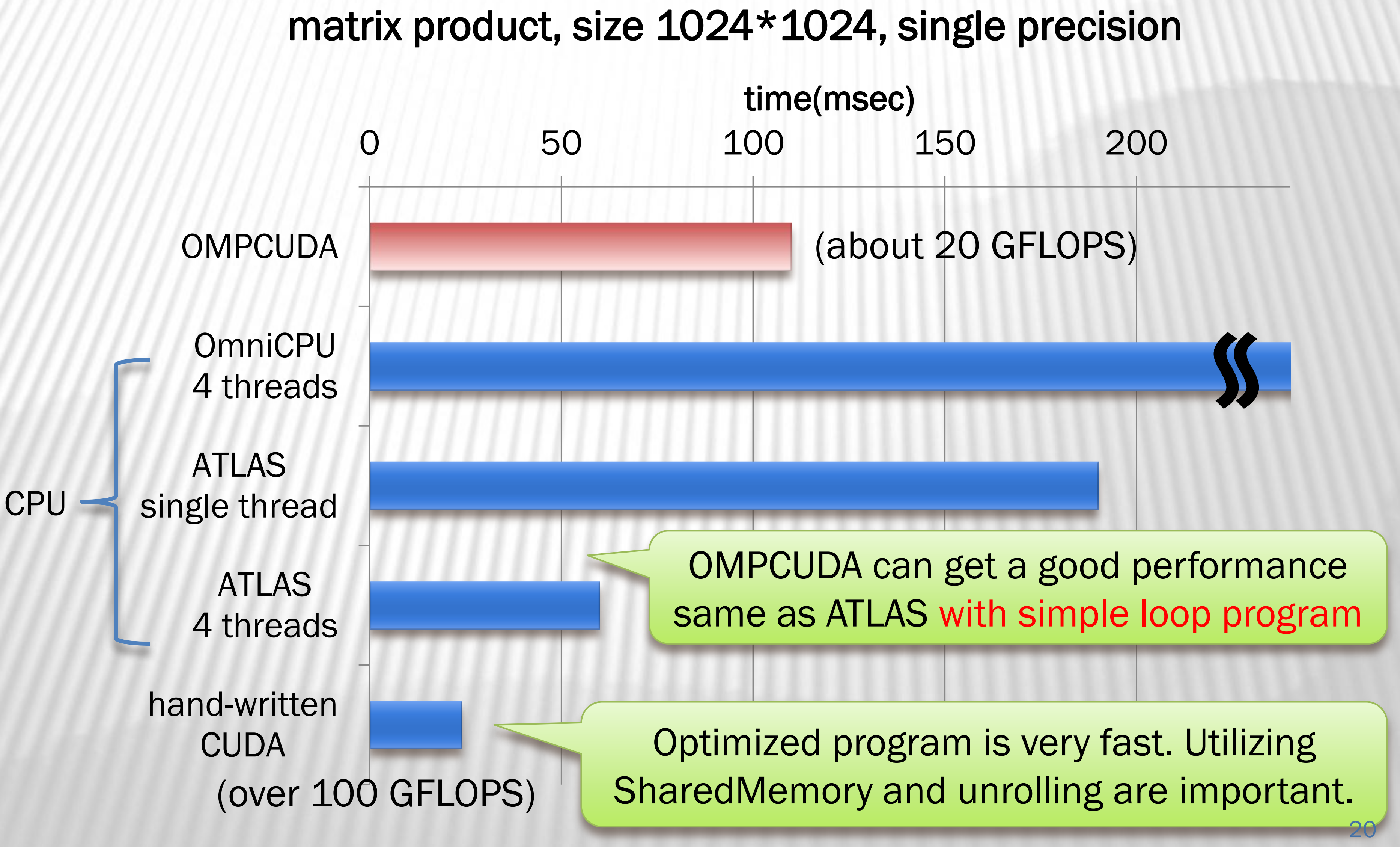
Matrix product

- Simple loop program
  - outer 2-fold loop is combined
  - (in order to enlarge the parallelism)

```
#define N 1024
float a[N*N], b[N*N], c[N*N];

#pragma omp parallel for private(j)
  for(i=0; i<N*N; i++){
    float tmp = 0.0f;
    for(j=0; j<N; j++){
      tmp += a[(i/N)*N+j] * b[j*N+(i%N)];
    }
    c[i] = tmp;
  }
```

Result of matrix product



OMPCUDA can get a good performance same as ATLAS with simple loop program

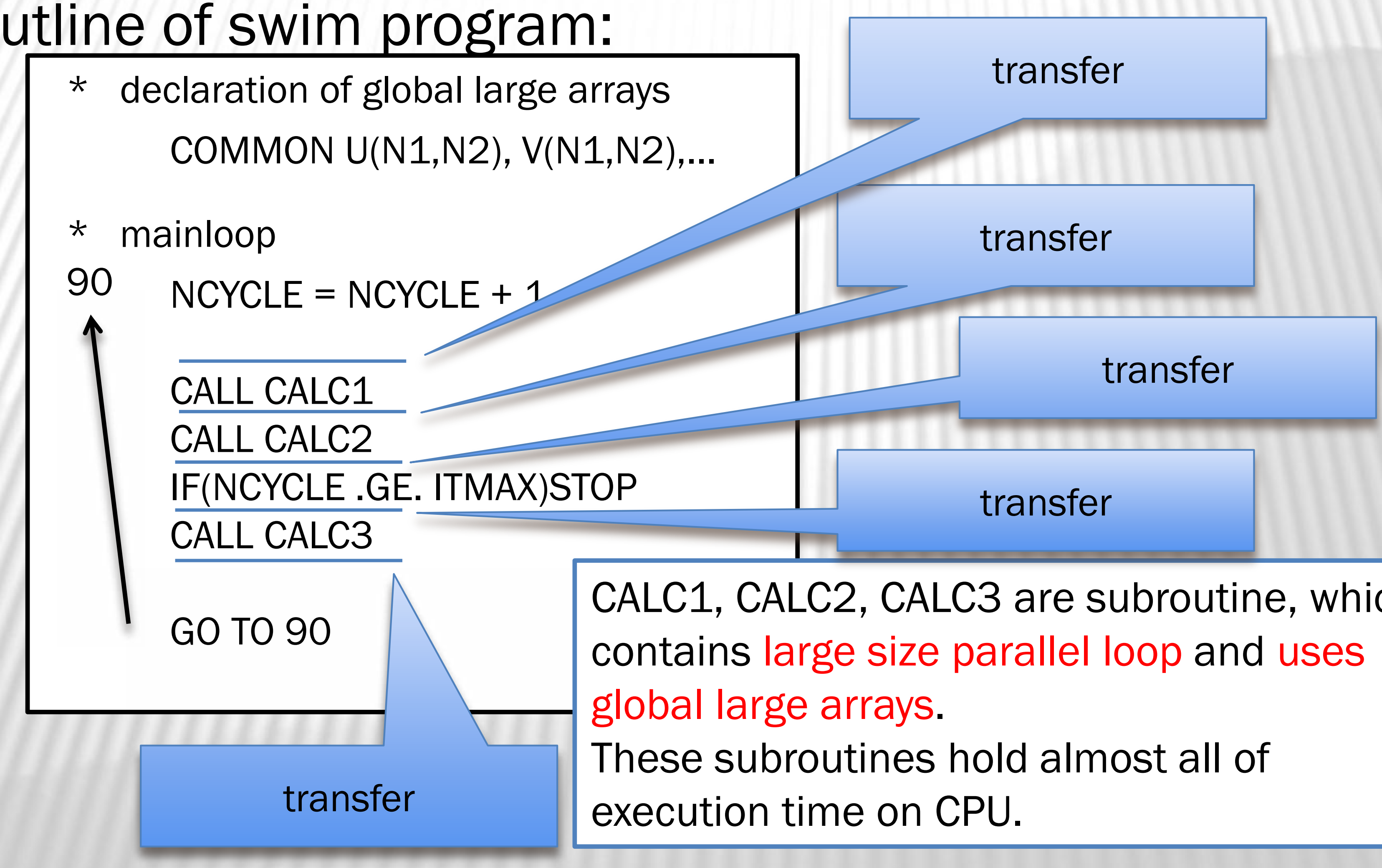
Optimized program is very fast. Utilizing SharedMemory and unrolling are important.

swim (SPEC OMP2001)

- swim
  - more realistic program than matrix product
  - one of the smallest and simplest program in SPEC OMP2001
  - double precision
  - only maximum size (constant number) is changed
- Performance (“test” dataset)
  - CPU with single thread: 0.2sec
  - OMPCUDA: 20sec
    - very slow, but it is not because of double precision

Why OMPCUDA get very low performance ?

outline of swim program:



```
*   declaration of global large arrays
        COMMON U(N1,N2), V(N1,N2),...

*   mainloop
90      NCYCLE = NCYCLE + 1

        CALL CALC1
        CALL CALC2
        IF(NCYCLE .GE. ITMAX)STOP
        CALL CALC3

        GO TO 90
```

transfer

transfer

transfer

transfer

transfer

CALC1, CALC2, CALC3 are subroutine, which contains large size parallel loop and uses global large arrays.

These subroutines hold almost all of execution time on CPU.

Room for improvement

- reduce the time of data transfer
  - leave the data on GPU
  - analyze program consistently using exist various techniques
- move data from GlobalMemory to SharedMemory and register
  - Can Fermi's cache memory solve this issue?
- other pragma
  - example: sections
    - assign to CUDA's Block level parallelization

Related Work

- Lee et al.*
  - OpenMP compiler for CUDA
    - has optimization mechanisms and has obtained high performance in some programs
    - We will be able to get their optimized technique.
- PGI
  - latest PGI compiler supports pragma-based parallel programming for CUDA in C/C++/Fortran
    - PGI's pragma is not equal to OpenMP pragma.
    - discussion: OpenMP pragma vs new pragma suitable for GPU

* Lee, S., Min, S.J., Eigenmann, R.: Openmp to gpgpu: a compiler framework for automatic translation and optimization. In: PPoPP '09, pp.101-110 (2009)

Conclusion

- “OMPCUDA”: We are developing OpenMP framework for CUDA.
- Motivation (Purpose)
  - Make GPU programming easy !
- Implementation
  - based on OMNI, we made program translator and runtime library
- Result
  - could get good performance by using normal OpenMP code
  - couldn’t get good performance in program with multiple kernels with large shared variables
- (many) Future work and challenges
  - corresponding to complex programs (pointer…)
  - cutting the transfer time (swim)
  - bringing in Lee’s technique
  - using SharedMemory (Can Fermi’s cache solve this?)
  - corresponding to Fortran90/95… (OMNI 1.6 only supports F77)

Thank you for your kind attention.

Question?

(ohshima@cc.u-tokyo.ac.jp)

acknowledgment: Omni Compiler project for releasing OMNI.