Review

- Simple Counting Example
  - Race condition
  - Atomic operation
  - Reduction

Today's Topic

- Vector reversal
  - Coalesced memory access
  - Using shared memory for coalescing memory access
  - Profiling
- Summing Matrix^2
  - Asynchronous Operations
  - Heterogeneous computing
  - Multiple devices

06?.cu

Vector Reversal

Vector reversal

```
void reverse(const int n, float *a, float *b) {
  for(int i=0;i<n;i++) {
    b[i] = a[n-i-1];
  }
}
```

Is this CPU-bound or memory-bound?

```
// simple/naïve version
__global__
void reverse_dev(const int n, float *a, float *b) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i<n) {
    b[i] = a[n-i-1];
  }
}
```


Coalescing memory access

- A coordinated memory access by "half-warp" (16 threads)
  - A warp is 32 threads – the minimal size of the threads processed by a multiprocessor.
  - Threads within a block are further organized into warps to be processed by one multiprocessor.
- A contiguous region of global memory:
  - 64 bytes: each thread reads a word: int, float, …
  - 128 bytes: each thread reads a double-word: int2, float2, …
  - 256 bytes: each thread reads a quad-word: int4, float4, …
- Additional restriction:
  - Starting address for a region must be a multiple of region size
  - The k-th thread in a half-warp must access the k-th element in a block being read
- Exception: not all threads need to participate

Naïve vector reversal

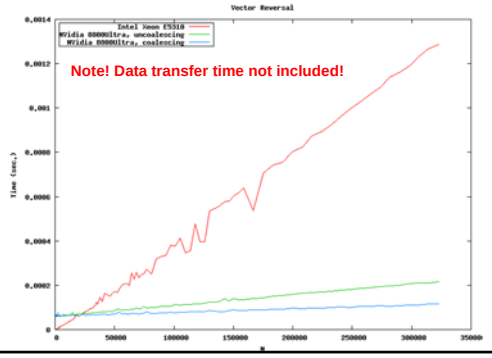
- Results in non-coalescing memory access on CUDA 1.0 and 1.1 device!
- Non-coalescing memory access results in reduced effective memory bandwidth.
- For compute capabilities >=1.2, the rules are much relaxed. (i.e. the simple version has good performance)
- Refer to Appendix F in CUDA_C Programming Guide (4.0rc2) for such information.

Vector reversal

```
// better version for CUDA 1.0 devices
__global__
void reverse_dev(const int n, float *a, float *b) {
  const int head = blockIdx.x * blockDim.x;
  __shared__ float smem[NX];
  const bool process = (head+threadIdx.x) < n;
  if(process) {
    float *from = a + n - head - blockDim.x;
    smem[blockDim.x-threadIdx.x-1]=from[threadIdx.x];
  }
  __syncthreads();
  if(process) {
    b[head+threadIdx.x] = smem[threadIdx.x];
  }
}
```

7

Performance



8

CUDA-C

new declspecs

- __shared__ float smem[NX];
  - *Declare a memory region that is shared by threads within the same block*
- Note shared memory usage have other issues known as bank conflict.
  - *Read Section 5.3.2.3 , Section F3.3 and F4.3 of the programming guide for more info.*

Variable declaration	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	Application

- *__device__ is optional when used with __local__, __shared__, or __constant__*
- *Automatic variables without any qualifier reside in a register, except arrays that reside in local memory.*

9

CUDA Profiling

export CUDA_PROFILE=1
export CUDA_PROFILE_CSV=1
export CUDA_PROFILE_LOG=06b.log
export CUDA_PROFILE_CONFIG=CUDA_PROFILE_CONFIG

Then run the program normally.
salloc -pCuda
srun./06b.exe 1000000
srun./06c.exe 1000000
exit

```
gld_incoherent
gst_incoherent
divergent_branch
warp_serialize
```

10

Profiling results

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce 8800 Ultra
# CUDA_PROFILE_CSV 1
# TIMESTAMPFACTOR fffff6f628f9b160
method,gputime,cputime,occupancy,gld_incoherent,gst_incoherent,divergent_branch,warp_serialize
memcpyHtoD,1297.856,1337.000
_Z11reverse_deviPfS_,575.520,614.000,1.000,125056,0,0,0
memcpyDtoH,1258.464,1270.000
```

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce 8800 Ultra
# CUDA_PROFILE_CSV 1
# TIMESTAMPFACTOR fffff6f6b209bab8
method,gputime,cputime,occupancy,gld_incoherent,gst_incoherent,divergent_branch,warp_serialize
memcpyHtoD,1297.824,1322.000
_Z11reverse_devmPfS_,222.176,274.000,1.000,0,0,0,0
memcpyDtoH,1257.568,1270.000
```

11

07?.cu

Summing Matrix^2

12

07a.cpp

```
void gemv(const size_t s, DT *a, DT *out) {
    for(size_t i=0;i<s;i++) {
        for(size_t j=0;j<s;j++) {
            for(size_t k=0;k<s;k++) {
                out[i*s+j] += a[i*s+k] * a[k*s+j];
            }
        }
    }
}
void addMatrices(const size_t n, const size_t s, DT
**a, DT *out) {
    for(size_t i=0;i<n;i++) {
        gemv(s, a[i], out);
    }
}
```

13

07b.cu

```
void addMatricesGPU(const size_t n, const size_t s, DT **a, DT
*out) {
  DT *tmp, *ans;
  size_t bytes = s*s*sizeof(DT);
  cudaMalloc(&tmp, bytes);
  cudaMalloc(&ans, bytes);
  cudaMemset(ans, 0, bytes);
  const int threads=16;
  dim3 blocks(threads, threads);
  dim3 grids((s+threads-1)/threads,(s+threads-1)/threads);
  for(size_t i=0;i<n;i++) {
    cudaMemcpy(tmp, a[i], bytes, cudaMemcpyHostToDevice);
    addKernel<<<grids, blocks>>>(s, tmp, ans);
  }
  cudaMemcpy(out, ans, bytes, cudaMemcpyDeviceToHost);
  cudaFree(tmp);
  cudaFree(ans);
}
```

```
for(size_t i=0;i<n;i++) {
    gemv(s, a[i], out);
}
```

14

07b.cu

```
__global__
void addKernel(size_t s, DT *a, DT *ans) {
  size_t i = blockIdx.y*blockDim.y + threadIdx.y;
  size_t j = blockIdx.x*blockDim.x + threadIdx.x;
  if(i<s && j<s) {
    for(size_t k=0;k<s;k++) {
      ans[i*s+j] += a[i*s+k] * a[k*s+j];
    }
  }
}
```

```
void gemv(const size_t s, DT *a, DT *out) {
  for(size_t i=0;i<s;i++) {
    for(size_t j=0;j<s;j++) {
      for(size_t k=0;k<s;k++) {
        out[i*s+j] += a[i*s+k] * a[k*s+j];
      }
    }
  }
}
```

Heterogeneous Computing

- All previous examples are done either using CPU or using GPU.
- When GPU does it work, CPU does nothing, and vice versa.
- Some CUDA APIs are non-blocking / asynchronous:
  - Kernel launches
  - Functions for copying memory and are suffixed with Async
  - Functions performing device ←→ device memory copies
  - Functions set memory (memset)
- CUDA introduced so called streams, which defines series of "commands" to be issued to CUDA devices. These commands are issued with very little delay to CPU (asynchronously, non-blocking).
- By using asynchronous APIs, we can make CPU and GPU work at the same time.

16

07c.cu

```
void addMatricesGPU(const size_t n, const size_t s, DT
**a, DT *out) {
  DT *tmp, *tmp1, *ans;
  size_t i, bytes = s*s*sizeof(DT);
  cudaMalloc(&tmp, bytes);
  cudaMalloc(&ans, bytes);
  cudaMallocHost(&tmp1, bytes);
```

17

07c.cu

```
// Define work to be done at GPU
const double ratio = 3.0/4;
const int threads=16;
dim3 blocks(threads, threads);
dim3 grids( (s+threads-1)/threads, (s+threads-1)/threads);
cudaStream_t stream;
cudaStreamCreate(&stream);
cudaMemsetAsync(ans, 0, bytes, stream);
for(i=0;i<n*ratio;i++) {
  cudaMemcpyAsync(tmp,a[i],bytes,cudaMemcpyHostToDevice,stream);
  addKernel<<<grids, blocks, 0, stream>>>(s, tmp, ans);
}
cudaMemcpyAsync(tmp1,ans,bytes,cudaMemcpyDeviceToHost,stream);

// CPU does rest of the work
for(;i<n;i++) {
  gemv(s, a[i], out);
}
```

18

07c.cu

```
  // Merge results
  cudaStreamSynchronize(stream);
  for(i=0;i<bytes;i++) {
    out[i] += tmp1[i];
  }

  cudaFree(tmp);
  cudaFree(ans);
  cudaFree(tmp1);
  cudaStreamDestroy(stream);
}
```

19

Asynchronous Operations

- Concurrent data transfer & kernel launch (limited)
  - Compute capability 1.1+
- Concurrent heterogeneous kernel launches
  - Compute capability 2.0+, 4 max
- Concurrent data transfers (uploads & downloads)
  - Compute capability 2.0+
- Streams
  - A sequence of commands that execute in order.
  - Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore not be relied upon for correctness (e.g. inter-kernel communication is undefined).
- Events
  - For timing purposes

20

Multiple CUDA Devices

- It is possible to use multiple CUDA devices at the same time on one host.
  - cudaGetDeviceCount(&deviceCount);
  - cudaSetDevice(device);

21

07d.cu

```
void addMatricesGPU(const size_t n, const size_t s, DT **a, DT
*out) {
  int deviceCount;
  cudaGetDeviceCount(&deviceCount);
  addMatricesGPUn(n, s, a, out, deviceCount);
}

void addMatricesGPUn(const size_t n, const size_t s, DT **a,
DT* const out, const int deviceCount=1) {
  DT *tmp[deviceCount], *ans[deviceCount];
  DT *tmpAns[deviceCount];
  const size_t bytes = s*s*sizeof(DT);
  const int threads=16;
  dim3 blocks(threads, threads);
  dim3 grids( (s+threads-1)/threads, (s+threads-1)/threads);
```

22

07d.cu

```
  omp_set_num_threads(deviceCount);
  #pragma omp parallel for
  for(int device=0;device<deviceCount;device++) {
    cudaSetDevice(device);
    if(device!=0) cudaMallocHost(&tmpAns[device], bytes);
    else          tmpAns[0] = out;
    cudaMalloc(&tmp[device], bytes);
    cudaMalloc(&ans[device], bytes);
    cudaMemset(ans[device], 0, bytes);
    for(size_t i=device;i<n;i+=deviceCount) {
      cudaMemcpy(tmp[device],a[i],bytes,cudaMemcpyHostToDevice);
      addKernel<<<grids, blocks>>>(s, tmp[device], ans[device]);
    }
    cudaMemcpy(tmpAns[device],ans[device],bytes, cudaMemcpyDeviceToHost);
    cudaFree(tmp[device]); cudaFree(ans[device]);
    cudaThreadSynchronize();
  }
```

23

07d.cu

```
  // merge results from multiple GPUs on CPU
  for(size_t device=1;device<deviceCount;device++) {
    for(size_t i=0;i<s*s;i++) {
      out[i] += tmpAns[device][i];
    }
    cudaFree(tmpAns[device]);
  }
}
```

24

Summary

- Devices → running multiple threads arranged in blocks, which are arranged in grid. The arrangement of blocks and grids replaces for-loop in conventional programming.
- Threads run kernels. GPU requires running multiple threads to be advantageous.
- Memory transfer operations are essential to perform useful tasks on GPU and obtain results calculated on GPU back to CPU for data storage and display.
- Need to learn to use different types of memory and need to be aware of efficient memory access patterns required for different devices.
- The last example uses OpenMP to generate multiple threads, each threads works with one CUDA device.

25

Lecture 12

26

Topics

- Introduction
- OpenMP
  - Compiler directives
  - Some Examples
  - Library functions
  - Environment variables

27

Introduction

- OpenMP is:
  - a standard for parallel programming in C, C++, and Fortran on shared memory computers.
  - portable, just like MPI.
  - widely supported across vendors, including Intel, Microsoft, HP, SGI, IBM, Sun, etc…
  - gaining importance due to the availability of **multi-core** CPUs.
- GCC 4.2, Intel Compiler 7.x, and VS2005 starts supporting OpenMP 2.0+. (VS2010: OpenMP 2.0 … Orz)
- OpenMP is now 3.0 (announced on 2008.05)
- OpenMP consists of:
  - Compiler directives (comments)
  - Library functions (function calls, API)
  - Environment variables

28

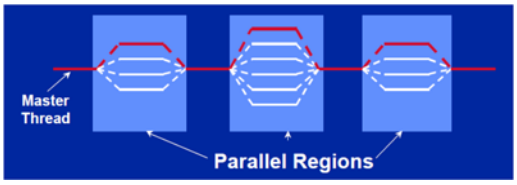
Shared Memory Parallelization

- Multithread programming
  - Thread: An execution entity having a serial flow of control and an associated stack.
- All processors can access all the memory in the parallel system (one address space).
- The time to access the memory may not be equal for all processors. (NUMA)
- Parallelizing on a SMP does not reduce CPU time.
  - It reduces wall-clock time.
- Parallel execution is achieved by generating multiple threads which execute in parallel.
- Number of threads (in principle) is independent of the number of processors

29

OpenMP Execution Model
Fork-Join model

- Master (or initial) thread spawns a team of threads as needed.
- Parallelism (team: master + workers) is added incrementally: i.e. the sequential program evolves into a parallel program.



Tim Mattson and Rudolf Eigenmann (1999), "OpenMP: An API for Writing Portable SMP Application Software"

30

Example

```
for(i=0;i<n;i++) {
    x[i]=2.0*x[i]+3.0*sqrt(x[i])-exp(x[i]);
}
```

```
#pragma omp parallel for
for(i=0;i<n;i++) {
    x[i]=2.0*x[i]+3.0*sqrt(x[i])-exp(x[i]);
}
```

Easy to create multithreads, and each thread can be scheduled by OS to different processors to take advantage of multiple CPU cores.

31

Syntax

- OpenMP consists of:
  - Compiler directives
  - *Library functions (function calls, API)*
  - *Environment variables*

#pragma omp *directive* [*clause* [*clause*] ...]

32

OpenMP

Compiler directives
Library functions
Environment variables

33

Compiler directives

Conditional compilation
Parallel construct
Work-sharing constructs
Work-tasking constructs
Synchronization

34

Compiler directives

conditional compilation

- A macro "**_OPENMP**" is defined with the value yyyymm
  - yyyy: year of the implemented OpenMP
  - mm: month of the implemented OpenMP

00.cpp

```
#include <iostream>

using namespace std;

int main() {
  #ifdef _OPENMP
    cout << "OpenMP: " << _OPENMP << endl;
  #else
    cout << "Your compiler does not support OpenMP." << endl;
  #endif
  return 0;
}
```

icpc -openmp 00.cpp
OpenMP: 200805

icpc 00.cpp
Your compiler does not support OpenMP.

35

Compiler directives

Conditional compilation
Parallel construct
Work-sharing constructs
Work-tasking constructs
Synchronization

36

Compiler directives
parallel construct

- A parallel construct defines the structured block followed being executed by a team of threads.

01a.cpp
```
#include <iostream>

using namespace std;

int main() {
  #pragma omp parallel
  {
    cout << "Hello World" << endl;
  }
  return 0;
}
```

A construct

```
./01a.exe
Hello World
Hello World
```

*Structured block: an executable statement, possibly compound, with a **single entry** at the top and a **single exit** at the bottom.*

37

Examples: structured block

```
cout <<"Hello World" << endl;
```

```
{
  a=a+b;
  c++;
  cout << "Hello World" << endl;
}
```

```
{
    cout << "Hello World" << endl;
    cout << "Hello Kitty" << endl;
    cout << "Hello Penguin" << endl;
}
```

```
for(i=0;i<100;i++) {
    cout << "Hello World" << endl;
    cout << "Hello Kitty" << endl;
    cout << "Hello Penguin" << endl;
}
```

38

Examples: non-structured block

```
{
  a=a+b;
  goto abc
  c++;
  cout << "Hello World" << endl;
}
:abc
```

```
for(i=0;i<100;i++) {
    cout << "Hello World" << endl;
    if(i==20) break;
    cout << "Hello Penguin" << endl;
}
```

39

Compiler directives
parallel construct

#pragma omp parallel *[clause [[,] clause] …] new-line*
structured-block

Clauses

num_threads(integer-expression)

private(list)
default(shared | none)
shared(list)
firstprivate(list)
**copyin (operator:list)*

reduction(operator: list)

if(scalar-expression)

OpenMP Application Program Interface
Section 2.9 Data Environment

40

Compiler directives
parallel construct

- num_threads(n): defines the number of threads to be created.
  - n: must be an integer expression (numbers, variables, …)

02a.cpp
```
#include <iostream>

using namespace std;

int main() {
  #pragma omp parallel num_threads(8)
  {
    cout << "Hello World" << endl;
  }
  return 0;
}
```

```
./01a.exe
Hello WorldHello World

Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

41

Compiler directives
parallel construct

- Data sharing: defines the sharing attribute of variables between threads in the parallel region
  - private(list): list the variables that are private in each thread
  - shared(list): list the variables that are shared between threads
  - default(shared|none): set the default sharing attribute of variables not listed.
- Private variables
  - Private variables are undefined on entry and exit of the parallel region.
  - The value of the original variable (before the parallel region) is **UNDEFINED** after the parallel region.
  - Variables declared in the parallel region are private variables.
  - parallel for 's index variables are private variables.
- Shared variables
  - These variables are shared between all threads
  - They cannot be declared in the structure block followed.

42

03a.cpp

```
#include <iostream>
using namespace std;
int main() {
  int count=17;
  #pragma omp parallel private(count)
  {
    count=0;
    for(int i=0;i<10;i++) {
      count++;
    }
    cout << "My count: " << count << endl;
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./03a.exe
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
Main count: 17
```

```
OMP_NUM_THREADS=8 ./03a.exe
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
Main count: 17
```

```
OMP_NUM_THREADS=8 ./03a.exe
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
Main count: 17
```

```
OMP_NUM_THREADS=8 ./03a.exe
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
Main count: 17
```

```
OMP_NUM_THREADS=8 ./03a.exe
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
Main count: 17
```

The variable count inside the parallel construct is in fact a new variable that has nothing to do with the count declared in the main().! 43

04a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int count=17;
  #pragma omp parallel shared(count)
  {
    for(int i=0;i<1000000;i++) {
      count++;
    }
    cout << "My count: " << count << endl;
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: My count:
10919331091933
My count: 2170209

My count: 3213347
My count: 3751776
My count: 4269572
My count: 4272164
My count: 4347902
Main count: 4347902
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1501215
My count: 2501215
My count: 3501215
My count: 4501215
My count: 5501215
My count: 6501215
My count: 7501215
My count: 1000017
Main count: 1000017
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1089747
My count: 2126479
My count: 2240935
My count: 3165806
My count: 3345987
My count: 3364074
My count: 4210268
My count: 4425132
Main count: 4425132
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1152417
My count: 2249624
My count: 2274868
My count: 3346454
My count: 3858762
My count: 4402852
My count: 4406426
My count: 4465075
Main count: 4465075
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1000017
My count: 2000017
My count: 3000017
My count: 4000017
My count: 5000017
My count: 6000017
My count: 7000017
My count: 8000017
Main count: 8000017
```

The variable count is now the one declared in the main(), and are shared by all threads. The show results are known as race condition. 44

05a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int count=17;
  #pragma omp parallel firstprivate(count)
  {
    for(int i=0;i<10;i++) {
      count++;
    }
    cout << "My count: " << count << endl;
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./05a.exe
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
Main count: 17
```

```
OMP_NUM_THREADS=8 ./05a.exe
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
Main count: 17
```

```
OMP_NUM_THREADS=8 ./05a.exe
My count: My count: 2727

My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
Main count: 17
```

```
OMP_NUM_THREADS=8 ./05a.exe
My count: My count: 2727
My count: 27
My count: 27
My count: 27

My count: 27
My count: 27
My count: 27
Main count: 17
```

```
OMP_NUM_THREADS=8 ./05a.exe
My count: My count: 2727
My count: 27
My count: 27
My count: 27

My count: 27
My count: 27
My count: 27
Main count: 17
```

Firstprivate "first" initializes newly created private variables within each thread with the original variable. (If the variable is an object, it is created using copy constructor). 45

06a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int count=17;
  #pragma omp parallel default(none)
  {
    for(int i=0;i<1000000;i++) {
      count++;
    }
    cout << "My count: " << count << endl;
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
icpc -O2 -Wall -openmp 06a.cpp -o 06a.exe stopWatch.o
06a.cpp(6): error: "count" must be specified in a variable list at
enclosing OpenMP parallel pragma
        #pragma omp parallel default(none)
        ^
06a.cpp(6): error: "std::cout" must be specified in a variable list at
enclosing OpenMP parallel pragma
        #pragma omp parallel default(none)
        ^
compilation aborted for 06a.cpp (code 2)
make: *** [06a.exe] Error 2
```

46

07a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int sum=0;
  #pragma omp parallel reduction(+:sum)
  {
    for(int i=0;i<=10;i++) {
      sum+=i;
    }
    cout << "My sum: " << sum << endl;
  }
  cout << "Main sum: " << sum << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./07a.exe
My sum: 55
My sum: 55
My sum: 55
My sum: 55
My sum: 55
My sum: 55
My sum: 55
My sum: 55
Main sum: 440
```

```
OMP_NUM_THREADS=4 ./07a.exe
My sum: 55
My sum: 55
My sum: 55
My sum: 55
Main sum: 220
```

```
OMP_NUM_THREADS=2 ./07a.exe
My sum: 55
My sum: 55
Main sum: 110
```

47

08a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int sum=0, n;

  cout << "\nEnter 0 to run single thread:";
  cin >> n;
  #pragma omp parallel reduction(+:sum) if(n)
  {
    for(int i=0;i<=10;i++) {
      sum+=i;
    }
    cout << "My sum: " << sum << endl;
  }
  cout << "Main sum: " << sum << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=4 ./08a.exe

Enter 0 to run single thread:1
My sum: 55
My sum: 55
My sum: 55
My sum: 55
Main sum: 220
```

```
OMP_NUM_THREADS=4 ./08a.exe

Enter 0 to run single thread:0
My sum: 55
Main sum: 55
```

48