Advanced Computing for Engineering Applications

Dan Negrut

Simulation-Based Engineering Lab
Wisconsin Applied Computing Center
Department of Mechanical Engineering
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
What We Covered Yesterday…

- The NVIDIA GPU memory ecosystem
- Execution scheduling
- Thread divergence
- Global memory access issues
- Atomics in CUDA
- Profiling CUDA executables
- General CUDA programming rules of thumb
What We’ll Be Covering Today

- CUDA GPU computing with Thrust
- Getting started with multi-core parallel computing as enabled by OpenMP
GPU Computing using **thrust**
3 Ways to Accelerate on GPU

Application

Libraries

Directives

Programming Languages

Easiest Approach

Maximum Performance

Direction of increased performance (and effort)

NVIDIA [C. Woolley]
Acknowledgments

- The **thrust** slides include material provided by Nathan Bell of NVIDIA
- Slightly modified, assuming responsibility for any mistakes
Design Philosophy, *thrust*

- Increase programmer productivity
  - Build complex applications quickly

- Adopt a generic programming angle
  - Leverage a template-based approach

- Should run fast
  - Efficient mapping to hardware
What is **thrust**?

- A template library for CUDA
  - Mimics the C++ STL

- Containers
  - On host and device

- Algorithms
  - Sorting, reduction, scan, etc.
What is **thrust**?

[Cntd.]

- **thrust** is a header library – all the functionality is accessed by `#include`-ing the appropriate `thrust` header file.

- Program is compiled with `nvcc` as per usual, no special tools are required.

- Lots of C++ syntax, related to high-level host-side code that you write.
  - The concept of execution configuration, shared memory, etc. : it’s all gone.
Example: Vector Addition

```
for (int i = 0; i < N; i++)
    Z[i] = X[i] + Y[i];
```
Example, Vector Addition

```cpp
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {
    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);


    thrust::transform(X.begin(), X.end(),
                      Y.begin(),
                      Z.begin(),
                      thrust::plus<float>())
                      ;

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << 

    return 0;
}
```

Example, Vector Addition

NVIDIA [N. Bell]→
Example, Vector Addition

[negrut@euler01 CodeBits]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Thu_Jan_12_14:41:45_PST_2012
Cuda compilation tools, release 4.1, V0.2.1221
[negrut@euler01 CodeBits]$ nvcc -O2 exThrust.cu -o exThrust.exe
[negrut@euler01 CodeBits]$ ./exThrust.exe
Z[0] = 25
Z[1] = 55
Z[2] = 40
[negrut@euler01 CodeBits]$
Example: SAXPY

```c
for (int i = 0; i < N; i++)
    Z[i] = a * X[i] + Y[i];
```
```cpp
struct saxpy
{
    float a;

    saxpy(float a) : a(a) {}

    __host__ __device__
    float operator()(float x, float y)
    {
        return a * x + y;
    }
};

int main(void)
{
    thrust::device_vector<float> X(3), Y(3), Z(3);


    float aVal = 2.0f;

    thrust::transform(X.begin(), X.end(),
                      Y.begin(), Z.begin(), saxpy(aVal));

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void) {
    // generate about 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
Containers

- Concise and readable code
  - Avoids common memory management errors
    - e.g.: Vectors automatically release memory when they go out of scope

```cpp
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// ... compute entries in h_vec ...

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// write device values
d_vec[0] = 13;
d_vec[1] = 27;

// read device values from the host
std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;
```
Containers

- Compatible with STL containers

```cpp
// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);

// copy list to device vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// alternative method using vector constructor
thrust::device_vector<int> d_vec2(h_list.begin(), h_list.end());
```
Namespaces

- Avoid name collisions

```cpp
// allocate host memory
thrust::host_vector<int> h_vec(10);

// call STL sort
std::sort(h_vec.begin(), h_vec.end());

// call Thrust sort
thrust::sort(h_vec.begin(), h_vec.end());

// for brevity
using namespace thrust;

// without namespace
int sum = reduce(h_vec.begin(), h_vec.end());
```
Algorithms

- Elementwise operations
  - for_each, transform, gather, scatter ...

- Reductions
  - reduce, inner_product, reduce_by_key ...

- Prefix Sums [scans]
  - inclusive_scan, inclusive_scan_by_key ...

- Sorting
  - sort, stable_sort, sort_by_key ...
# Thrust Example: Sort

```c++
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void) {
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (805 Mkeys/sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

NVIDIA [N. Bell]→
Leveraging Parallel Primitives

- Test: sort 32M keys on each platform
  - Performance measured in millions of keys per second [higher is better]
- Conclusion: Use `sort` liberally, it’s highly optimized

<table>
<thead>
<tr>
<th>data type</th>
<th><code>std::sort</code></th>
<th><code>tbb::parallel_sort</code></th>
<th><code>thrust::sort</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>25.1</td>
<td>68.3</td>
<td>3532.2</td>
</tr>
<tr>
<td>short</td>
<td>15.1</td>
<td>46.8</td>
<td>1741.6</td>
</tr>
<tr>
<td>int</td>
<td>10.6</td>
<td>35.1</td>
<td>804.8</td>
</tr>
<tr>
<td>long</td>
<td>10.3</td>
<td>34.5</td>
<td>291.4</td>
</tr>
<tr>
<td>float</td>
<td>8.7</td>
<td>28.4</td>
<td>819.8</td>
</tr>
<tr>
<td>double</td>
<td>8.5</td>
<td>28.2</td>
<td>358.9</td>
</tr>
</tbody>
</table>
Input-Sensitive Optimizations

![Graph showing Sorting Rate (Mkey/s) vs Key Bits]

NVIDIA [N. Bell]
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {
    thrust::device_vector<float> X(3);


    float init = X[0];

    float result = thrust::reduce(X.begin(), X.end(),
                                   init,
                                   thrust::maximum<float>());

    std::cout << "maximum is " << result << "\n";

    return 0;
}
Algorithms

- Process one or more ranges

```cpp
// copy values to device
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// sort A in-place
sort(A.begin(), A.end());

// copy A -> B
copy(A.begin(), A.end(), B.begin());

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());
```
Algorithms

- Standard operators

```cpp
// allocate memory
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());

// transform A - B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), minus<int>());

// multiply reduction
int product = reduce(A.begin(), A.end(), 1, multiplies<int>());
```
Algorithms

- Standard data types

```cpp
// allocate device memory
device_vector<int> i_vec = ...
device_vector<float> f_vec = ...

// sum of integers
int i_sum = reduce(i_vec.begin(), i_vec.end());

// sum of floats
float f_sum = reduce(f_vec.begin(), f_vec.end());
```
Custom Types & Operators

```cpp
struct negate_float2
{
    __host__ __device__
    float2 operator()(float2 a)
    {
        return make_float2(-a.x, -a.y);
    }
};

// declare storage
device_vector<float2> input = ...
device_vector<float2> output = ...

// create function object or ‘functor’
negate_float2 func;

// negate vectors
transform(input.begin(), input.end(), output.begin(), func);
```
// compare x component of two float2 structures
struct compare_float2
{
    __host__ __device__
    bool operator()(float2 a, float2 b)
    {
        return a.x < b.x;
    }
};

// declare storage
device_vector<float2> vec = ...;

// create comparison functor
compare_float2 comp;

// sort elements by x component
sort(vec.begin(), vec.end(), comp);
// return true if x is greater than threshold
struct is_greater_than
{
    int threshold;

    is_greater_than(int t) { threshold = t; }

    __host__ __device__
    bool operator()(int x) { return x > threshold; }
};

device_vector<int> vec = ...

// create predicate functor (returns true for x > 10)
is_greater_than pred(10);

// count number of values > 10
int result = count_if(vec.begin(), vec.end(), pred);
Interoperability

- Convert iterators to raw pointers

```cpp
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector’s memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<< N / 256, 256 >>>(N, ptr);

// use ptr in a CUDA API function
cudaMemcpyAsync(ptr, ...);
```
Interoperability

- Wrap raw pointers with `device_ptr`

```c
// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```
## Algorithms, More Context…

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>Sum of a sequence</td>
</tr>
<tr>
<td>find</td>
<td>First position of a value in a sequence</td>
</tr>
<tr>
<td>mismatch</td>
<td>First position where two sequences differ</td>
</tr>
<tr>
<td>inner_product</td>
<td>Dot product of two sequences</td>
</tr>
<tr>
<td>equal</td>
<td>Whether two sequences are equal</td>
</tr>
<tr>
<td>min_element</td>
<td>Position of the smallest value</td>
</tr>
<tr>
<td>count</td>
<td>Number of instances of a value</td>
</tr>
<tr>
<td>is_sorted</td>
<td>Whether sequence is in sorted order</td>
</tr>
<tr>
<td>transform_reduce</td>
<td>Sum of transformed sequence</td>
</tr>
</tbody>
</table>
General Transformations

Unary Transformation

\[
\text{for (int } i = 0; i < N; i++)
X[i] = f(A[i]);
\]

Binary Transformation

\[
\text{for (int } i = 0; i < N; i++)
X[i] = f(A[i], B[i]);
\]

Ternary Transformation

\[
\text{for (int } i = 0; i < N; i++)
X[i] = f(A[i], B[i], C[i]);
\]

General Transformation

\[
\text{for (int } i = 0; i < N; i++)
X[i] = f(A[i], B[i], C[i], \ldots);
\]

- Like C++ STL, thrust provides built-in support for unary and binary transformations
- Transformations involving 3 or more input ranges must use a different approach
General Transformations Preamble:
The Zipping Operation

Multiple Distinct Sequences

Unique Sequence of Tuples

zip_iterator
Example: General Transformations

```cpp
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include<iostream>

struct linear_combo {
  __host__ __device__
  float operator()(thrust::tuple<float,float,float> t) {
    float x, y, z;
    thrust::tie(x,y,z) = t;
    return 2.0f * x + 3.0f * y + 4.0f * z;
  }
};

int main(void) {
  thrust::device_vector<float> X(3), Y(3), Z(3);
  thrust::device_vector<float> U(3);


  thrust::transform
    (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
     thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
     U.begin(), linear_combo());

  for (size_t i = 0; i < Z.size(); i++)
    std::cout << "U[" << i << "] = " << U[i] << "\n";
  return 0;
}
```

Functor Definition

These are the important parts: three different entities are zipped together in one big one

Example: General Transformations
```cpp
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__ float operator()(thrust::tuple<float, float, float> t) {
        float x, y, z;
        thrust::tie(x, y, z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3), U(3);

    thrust::plus<float> binary_op;
    float init = 0.f;

    float myResult = thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         linear_combo(),
         init,
         binary_op);

    std::cout << myResult << std::endl;
    return 0;
}
```
thrust, Efficiency Issues
[fusing transformations]
Performance Considerations
[short detour: 1/3]

- Picture below shows key parameters
  - Peak flop rate
  - Max bandwidth

---

1030 GFLOP/s [SinglePrecision]

SMs 144 GB/s

Tesla C2050

NVIDIA [N. Bell]→
Arithmetic Intensity
[short detour: 2/3]

- SAXPY
- FFT
- SGEMM

FLOP/Byte

Memory bound

Compute bound

NVIDIA [N. Bell]→
## Arithmetic Intensity

### Kernel FLOP/Byte*

<table>
<thead>
<tr>
<th>Kernel</th>
<th>FLOP/Byte*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Addition</td>
<td>1 : 12</td>
</tr>
<tr>
<td>SAXPY</td>
<td>2 : 12</td>
</tr>
<tr>
<td>Ternary Transformation</td>
<td>5 : 20</td>
</tr>
<tr>
<td>Sum</td>
<td>1 : 4</td>
</tr>
<tr>
<td>Max Index</td>
<td>1 : 12</td>
</tr>
</tbody>
</table>

* excludes indexing overhead

### Hardware**

<table>
<thead>
<tr>
<th>Hardware</th>
<th>FLOP/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX 280</td>
<td>~7.0 : 1</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>~7.6 : 1</td>
</tr>
<tr>
<td>Tesla C870</td>
<td>~6.7 : 1</td>
</tr>
<tr>
<td>Tesla C1060</td>
<td>~9.1 : 1</td>
</tr>
<tr>
<td>Tesla C2050</td>
<td>~7.1 : 1</td>
</tr>
</tbody>
</table>

** lists the number of flop per byte of data to reach peak Flop/s rate

“Byte” refers to a Global Memory byte
Fusing Transformations

\[
\text{for (int } i = 0; i < N; i++)
\]
\[
U[i] = F(X[i], Y[i], Z[i]);
\]
\[
\text{for (int } i = 0; i < N; i++)
\]
\[
V[i] = G(X[i], Y[i], Z[i]);
\]

Loop Fusion

- One way to look at things…
  - Zipping: reorganizes data for thrust processing
  - Fusing: reorganizes computation for efficient thrust processing

NVIDIA [N. Bell]→
typedef thrust::tuple<float,float> Tuple2;
typedef thrust::tuple<float,float,float> Tuple3;

struct linear_combo {
    __host__ __device__
    Tuple2 operator()(Tuple3 t) {
        float x, y, z; thrust::tie(x,y,z) = t;
        float u = 2.0f * x + 3.0f * y + 4.0f * z;
        float v = 1.0f * x + 2.0f * y + 3.0f * z;
        return Tuple2(u,v);
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3), V(3);
    thrust::transform
    (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
     thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
     thrust::make_zip_iterator(thrust::make_tuple(U.begin(), V.begin())),
     linear_combo());
    return 0;
}
Since the operation is completely memory bound the expected speedup is $\sim 1.6x (=32/20)$
Fusing Transformations

```c
for (int i = 0; i < N; i++)
    Y[i] = F(X[i]);

for (int i = 0; i < N; i++)
    sum += Y[i];

for (int i = 0; i < N; i++)
    sum += F(X[i]);
```

Loop Fusion
#include <thrust/device_vector.h>
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

int main(void) {
    thrust::device_vector<float> X(3);


    float result = thrust::transform_reduce
        (X.begin(), X.end(),
         _1 * _1,
         0.0f,
         thrust::plus<float>())

    std::cout << "sum of squares is " << result << "\n";
    return 0;
}
Fusing Transformations

Original Implementation

- GPU
- 4 Bytes → DRAM
- 4 Bytes → DRAM
- 4 Bytes → DRAM

Optimized Implementation

- GPU
- 4 Bytes → DRAM

Try to answer this: how many times will we be able to run faster if we fuse?
typedef thrust::tuple<int,int> Tuple;

struct max_index {
    __host__ __device__
    Tuple operator()(Tuple a, Tuple b) {
        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }
};

int main(void) {
    thrust::device_vector<int> X(3), Y(3);

    X[0] = 10; X[1] = 30; X[2] = 20; // values
    Y[0] = 0; Y[1] = 1; Y[2] = 2;  // indices

    Tuple init(X[0],Y[0]);

    Tuple result = thrust::reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end())),
         init,
         max_index());

    int value, index; thrust::tie(value,index) = result;

    std::cout << "maximum value is " << value << " at index " << index << "\n";

    return 0;
}
```cpp
typedef thrust::tuple<int,int> Tuple;

struct max_index {
    __host__ __device__
    Tuple operator()(Tuple a, Tuple b) {
        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }
};

int main(void) {
    thrust::device_vector<int> X(3);
    thrust::counting_iterator<int> Y(0);
    Tuple init(X[0],Y[0]);
    Tuple result = thrust::reduce
                    (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y)),
                    thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y + X.size())),
                    init,
                    max_index());
    int value, index; thrust::tie(value,index) = result;
    std::cout << "maximum value is " << value << " at index " << index << "\n";
    return 0;
}
```

**Maximum Index** [better approach]
Maximum Index (Optimized)

Original Implementation

![Original Implementation Diagram]

- Try to answer this: how many times will we be able to run faster if we fuse?

Optimized Implementation

![Optimized Implementation Diagram]
Good Speedups Compared to Multi-threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

NVIDIA [N. Bell]→
thrust Wrap-Up

- Significant boost in productivity at the price of small performance penalty
  - No need to be aware of execution configuration, shared memory, etc.

- Key concepts
  - Functor
  - Zipping data
  - Fusing operations

- Why not always use thrust?
  - There is no “solve multibody dynamics” support in thrust.
    - I need to take care of this
  - Thrust provides support for primitives – up to us to use them as needed
thrust on Google Code

- Quick Start Guide
- Examples
- News
- Documentation
- Mailing List (thrust-users)
thrust in “GPU Computing Gems”

This chapter demonstrates how to leverage the Thrust parallel template library to implement high-performance applications with minimal programming effort. Based on the C++ Standard Template Library (STL), Thrust brings a familiar high-level interface to the realm of GPU Computing while remaining fully interoperable with the rest of the CUDA software ecosystem. Applications written with Thrust are concise, readable, and efficient.

26.1 Motivation

With the introduction of CUDA C/C++, developers can harness the massive parallelism of the GPU through a standard programming language. CUDA allows developers to make fine-grained decisions about how computations are decomposed into parallel threads and executed on the device. The level of control offered by CUDA C/C++ (henceforth CUDA C) is an important feature: it facilitates the development of high-performance algorithms for a variety of computationally demanding tasks which (1) merit significant optimization and (2) profit from low-level control of the mapping onto hardware.

For this class of computational tasks CUDA C is an excellent solution. Thrust (1) solves a complementary set of problems, namely those that are (1) implemented efficiently without a detailed mapping of work onto the target architecture or those that (2) do not merit or simply will not receive significant optimization effort by the user. With Thrust, developers describe their computation using a collection of high-level algorithms and completely delegate the decision of how to implement the computation to the library. This abstract interface allows programmers to describe what to compute without placing any additional restrictions on how to carry out the computation. By capturing the programmer’s intent at a high level, Thrust has the discretion to make informed

PDF available at http://goo.gl/adj9S
Example, **thrust**: Processing Rainfall Data

Rain situation, end of first day, for a set of five observation stations. Results, summarized over a period of time, reported in the table below.

| day | [0 0 1 2 5 5 6 6 7 8 ... ] |
| site | [2 3 0 1 1 2 0 1 2 1 ... ] |
| measurement | [9 5 6 3 3 8 2 6 5 10 ... ] |

Remarks:
1) Time series sorted by day
2) Measurements of zero are excluded from the time series
Example: Processing Rainfall Data

Given the data above, here’re some questions you might ask:

- Total rainfall at a given site
- Total rainfall between given days
- Total rainfall on each day
- Number of days with any rainfall
Total Rainfall at a Given Site

```cpp
struct one_site_measurement {
    int siteOfInterest;

    one_site_measurement(int site) : siteOfInterest(site) {} 

    __host__ __device__
    int operator()(thrust::tuple<int, int> t) {
        if (thrust::get<0>(t) == siteOfInterest)
            return thrust::get<1>(t);
        else
            return 0;
    }
};

template <typename Vector>
int compute_total_rainfall_at_one_site(int siteID, const Vector& site, const Vector& measurement) {
    return thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(site.begin(), measurement.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(site.end(), measurement.end())),
         one_site_measurement(siteID),
         0,
         thrust::plus<int>())
};
```
Total Rainfall Between Given Days

```cpp
template <typename Vector>
int compute_total_rainfall_between_days(int first_day, int last_day,
    const Vector& day, const Vector& measurement)
{
    int first = thrust::lower_bound(day.begin(), day.end(), first_day) - day.begin();
    int last = thrust::upper_bound(day.begin(), day.end(), last_day) - day.begin();

    return thrust::reduce(measurement.begin() + first, measurement.begin() + last);
}
```

For this to fly, you’ll need to include several header files (not all for the code snippet above):

```cpp
#include <thrust/device_vector.h>
#include <thrust/binary_search.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>
```

NVIDIA [N. Bell]→
Number of Days with Any Rainfall

template <typename Vector>
int compute_number_of_days_with_rainfall(const Vector& day)
{
    return thrust::inner_product(day.begin(), day.end() - 1, 
                                day.begin() + 1, 
                                0, 
                                thrust::plus<int>(), 
                                thrust::not_equal_to<int>()) + 1;
}
Total Rainfall on Each Day

```
template <typename Vector>
void compute_total_rainfall_per_day(const Vector& day, const Vector& measurement, Vector& day_output, Vector& measurement_output)
{
    size_t N = compute_number_of_days_with_rainfall(day); //see previous slide

day_output.resize(N);
measurement_output.resize(N);

    thrust::reduce_by_key(day.begin(), day.end(),
        measurement.begin(),
        day_output.begin(),
        measurement_output.begin());
}
```

<table>
<thead>
<tr>
<th>day</th>
<th>0 1 2 5 6 7 8 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 5 5 6 7 8 ...</td>
<td></td>
</tr>
</tbody>
</table>

| measurement | 9 5 6 3 8 2 6 5 10 ... |

<table>
<thead>
<tr>
<th>day_output</th>
<th>0 1 2 5 6 7 8 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 6 3 11 8 5 10 ...</td>
<td></td>
</tr>
</tbody>
</table>

| measurement_output | 14 6 3 11 8 5 10 ... |
3 Ways to Accelerate on GPU

Application

Libraries
Directives
Programming Languages

Easiest Approach
Maximum Performance

Direction of increased performance (and effort)
Directives…
OpenACC

- Seeks to become:
  - A standard for directives-based Parallel Programming
  - Provide portability across hardware platforms and compiler vendors
- Promoted by NVIDIA, Cray, CAPS, PGI
OpenACC Specification

- Hardware agnostic and platform independent (CPU only, different GPUs)

- OpenACC is an open standard for directives based computing

- Announced at SC11 [November 2011]

- Caps, Cray, and PGI shipping OpenACC Compilers as of Q1 2012
  - AMD and NVIDIA also promoting use of OpenACC

- A version 2.0 draft is out for comments, idea is to expand OpenMP with features that enables one to farm out work to the GPUs
The OpenACC Idea

- Host code computes an approximation for $\pi$:

```cpp
#include <iostream>
#include <math.h>
using namespace std;

int main( int argc, char *argv[] )
{
    const double PI25DT = 3.141592653589793;

    const int n=1000000;
    double h = 1.0 / (double) n;
    double sum = 0.0;

    for( int i=0; i<=n; i++ ) {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    double mypi = h * sum;

    cout << "Approx. value: " << mypi << endl;
    cout << "Error: " << fabs(mypi-PI25DT) << endl;
    return 0;
}
```
The OpenACC Idea

- Code computes an approximation for \( \pi \) [might use multi-core or GPU]

```cpp
#include <iostream>
#include <math.h>
using namespace std;

int main( int argc, char *argv[] )
{
    const double PI25DT = 3.141592653589793238462643;

    const int n=1000000;
    double h = 1.0 / (double) n;
    double sum = 0.0;
    // #pragma acc region for
    for( int i=0; i<=n; i++ ) {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    double mypi = h * sum;

    cout << "Approx. value: " << mypi << endl;
    cout << "Error: " << fabs(mypi-PI25DT) << endl;
    return 0;
}
```

Add one line of code (a directive): provides a hint to the compiler about opportunity for parallelism
OpenACC Target Audience

- OpenACC targets three classes of users:
  - Users with parallel codes, ideally with some OpenMP experience, but less GPU knowledge
  - Users with serial codes looking for portable parallel performance with and without GPUs
  - “Hardcore” GPU programmers with existing CUDA ports
OpenACC Perceived Benefits

- Code easier to maintain
- Helps with legacy code bases
- Portable:
  - Can run same code CPU/GPU
- Very much like OpenMP
- Only small performance loss
  - Cray goal: 90% of CUDA
CUDA: Getting More Info…

- More information on this
  

- CUDA Tools and Ecosystem
  - Described in detail on NVIDIA Developer Zone
    
    http://developer.nvidia.com/category/zone/cuda-zone
GPU Computing with CUDA: Wrapping Up

- First question you need to ask: is there a GPU library that I can use?

- If no library available, just do it!
  - Avoid “analysis paralysis”
  - Adopt a “crawl – walk – run” approach
  - Go back and profile/optimize once you have something working
  - To “have something working” debug like a pro (cuda-gdb and cuda-memchk)
Libraries...
CUDA Libraries

- Math, Numerics, Statistics
- Dense & Sparse Linear Algebra
- Algorithms (sort, etc.)
- Image Processing
- Signal Processing
- Finance

- In addition to these widely adopted libraries, several less established ones available in the community

cuBLAS: Dense linear algebra on GPUs

- Complete BLAS implementation plus useful extensions
  - Supports all 152 standard routines for single, double, complex, and double complex
  - Levels 1, 2, and 3 BLAS
Speedups Compared to Multi-threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuSPARSE: Sparse linear algebra routines

- Sparse matrix-vector multiplication & triangular solve
  - APIs optimized for iterative methods

- New features in 4.1:
  - Tri-diagonal solver with speedups up to 10x over Intel MKL
  - ELL-HYB format offers 2x faster matrix-vector multiplication

\[
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4 \\
\end{bmatrix} = \alpha \begin{bmatrix}
  2 & -1 \\
  4 & -1 & 2 \\
  5 & 9 & 1 & 1 \\
  -1 & 8 & 3 & 2 \\
\end{bmatrix} + \beta \begin{bmatrix}
  2 \\
  0 \\
  -1 \\
  2 \\
\end{bmatrix}
\]
Good Speedups Compared to Multi-threaded CPU Execution

Sparse matrix test cases on following slides come from:
1. The University of Florida Sparse Matrix Collection
   http://www.cise.ufl.edu/research/sparse/matrices/
   http://www.nvidia.com/object/nvidia_research_pub_001.html

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuFFT: Multi-dimensional FFTs

- Algorithms based on Cooley-Tukey and Bluestein
- Simple interface, similar to FFTW
- Streamed asynchronous execution
- 1D, 2D, 3D transforms of complex and real data
- Double precision (DP) transforms
- 1D transform sizes up to 128 million elements
- Batch execution for doing multiple transforms
- In-place and out-of-place transforms

\[
F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi \frac{x \cdot n}{N}}
\]

\[
f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x) e^{j2\pi \frac{x \cdot n}{N}}
\]
Speedups Compared to Multi-Threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

NVIDIA [C. Woolley]→
cuRAND: Random Number Generation

- Pseudo- and Quasi-RNGs
  - Supports several output distributions
  - Statistical test results reported in documentation

- New RNGs in CUDA 4.1:
  - MRG32k3a RNG
  - MTGP11213 Mersenne Twister RNG
NPP: NVIDIA Performance Primitives

- Arithmetic, Logic, Conversions, Filters, Statistics, Signal Processing, etc.
- This is where GPU computing shines
- 1,000+ new image primitives in 4.1
Development, Debugging, and Deployment Tools
[Rounding Up the CUDA Ecosystem]
Programming Languages & APIs

- HMPP Compiler
- Python for CUDA
- NVIDIA C Compiler
- CUDA Fortran
- PGI Accelerator
- CUDA-x86
- OpenCL
- DirectX 11
- Microsoft AMP C/C++
Debugging Tools

- NVIDIA Parallel Nsight for Visual Studio
- NVIDIA CUDA-MEMCHECK for Linux & Mac
- Allinea DDT with CUDA Distributed Debugging Tool
- NVIDIA CUDA-GDB for Linux & Mac
- TotalView for CUDA for Linux Clusters
Performance Analysis Tools

- NVIDIA Parallel Nsight for Visual Studio
- Vampir Trace Collector
- TAU Performance System
- Performance API Library
- NVIDIA Visual Profiler for Linux & Mac
- Under Development
MPI & CUDA Support

Peer-Peer Transfers

GPUDirect™

InfiniBand

As of OFED 1.5.2
Announced pre-release at SC2011
Platform MPI
Announced beta at SC2011

NVIDIA [C. Woolley]→
Cluster Management & Job Scheduling

- Platform Computing
  - LSF, HPC, Cluster Manager
- Bright Computing
  - Bright Cluster Manager
- Adaptive Computing
- PBS Works
  - PBS Professional
- Ganglia
  - NVML Plugin for GPUs
- UNIVA
  - Univa Grid Engine

C. Woolley
Multicore Parallel Computing with OpenMP
Quick Look at Hardware

- Intel Haswell
  - Released in June 2013
  - 22 nm technology
  - Transistor budget: 1.4 billions
    - Tri-gate, 3D transistors
  - Typically comes in four cores
  - Has an integrated GPU
  - Deep pipeline – 16 stages
  - Sophisticated infrastructure for ILP acceleration
  - Superscalar
  - Supports HTT (hyper-threading technology)

Good source of information for these slides: [http://www.realworldtech.com/](http://www.realworldtech.com/)
Quick Look at Hardware

- Actual layout of the chip

- Schematic of the chip organization
  - LLC: last level cache (L3)
  - Three clocks:
    - A core’s clock ticks at 2.7 to 3.0 GHz but adjustable up to 3.7-3.9 GHz
    - Graphics processor ticking at 400 MHz but adjustable up to 1.3 GHz
    - Ring bus and the shared L3 cache - a frequency that is close to but not necessarily identical to that of the cores
Caches

- **Data:**
  - L1 – 32 KB per core
  - L2 – 512 KB or 1024 KB per core
  - L3 – 8 MB per CPU

- **Instruction:**
  - L0 – room for about 1500 microoperations (uops) per core
    - See H/S primer, online
  - L1 – 32 KB per core

- **Cache is a black hole for transistors**
  - Example: 8 MB of L3 translates into:
    - $8 \times 1024 \times 1024 \times 8 \text{ (bits)} \times 6 \text{ (transistors per bit, SRAM)} = 402 \text{ million transistors out of 1.4 billions}$

- **Caches are *very* important for good performance**
Fermi Specifics

- There are two schedulers that issue warps of “ready-to-go” threads
- One warp issued at each clock cycle by each scheduler
- During no cycle can more than 2 warps be dispatched for execution on the four functional units
- Scoreboarding is used to figure out which warp is ready
Haswell Microarchitecture [30,000 Feet]

- Microarchitecture components:
  - Instruction pre-fetch support (purple)
  - Instruction decoding support (orange)
    - CISC into uops
      - Turning CISC to RISC
  - Instruction Scheduling support (yellowish)
  - Instruction execution
    - Arithmetic (blue)
    - Memory related (green)

- More details - see primer posted online:
  [http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf]
Moving from HW to SW
Acknowledgements

- Majority of slides used for discussing OpenMP issues are from Intel’s library of presentations for promoting OpenMP
  - Slides used herein with permission

- Credit given where due: IOMPP
  - IOMPP stands for “Intel OpenMP Presentation”
OpenMP and Symmetric Multi-Processing

- Threads have access to large pool of shared memory
- Threads can have private data
  - Not accessible by other threads
- Data transfer/access transparent to programmer
- Synchronization is implicit but can be made explicit as well
Data vs. Task Parallelism

- Data parallelism
  - You have a large amount of data elements and each data element needs to be processed to produce a result
  - When this processing can be done in parallel, we have data parallelism
  - Example:
    - Adding two long arrays of doubles to produce yet another array of doubles

- Task parallelism
  - You have a collection of tasks that need to be completed
  - If these tasks can be performed in parallel you are faced with a task parallel job
  - Examples:
    - Reading the newspaper, whistling a tune, and slowly scratching your back
Objectives

- Understand OpenMP at the level where you can
  - Implement data parallelism
  - Implement task parallelism
Work Plan: What We’ll Cover

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization
- Advanced topics
OpenMP: Target Hardware

- CUDA: targeted parallelism on the GPU

- OpenMP: targets parallelism on SMP architectures
  - Handy when
    - You have a machine that has 64 cores
    - You have a large amount of shared memory, say 128GB

- MPI: targeted parallelism on a cluster (distributed computing)
  - Note that MPI implementation can handle transparently an SMP architecture such as a workstation with two hexcore CPUs that draw on a good amount of shared memory
OpenMP: What’s Reasonable to Expect

- If you have 64 cores available to you, it is *highly* unlikely to get a speedup of more than 64 (superlinear)

- Recall the trick that helped the GPU hide latency
  - Overcommitting the SMs and hiding memory access latency with warp execution

- This mechanism of hiding latency by overcommitment does not *explicitly* exist for parallel computing under OpenMP beyond what’s offered by HTT
  - It exists implicitly, under the hood, through ILP support
OpenMP: What Is It?

- Portable, shared-memory threading API
  - Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows

- Standardizes task & loop-level parallelism
- Very good at coarse-grained parallelism
- Combines serial and parallel code in single source
- Standardizes ~ 25 years of compiler-directed threading experience

- Current spec is OpenMP 4.0
  - Released in 2013
  - [http://www.openmp.org](http://www.openmp.org)
  - More than 300 Pages
OpenMP Programming Model

- **Master thread** spawns a team of threads as needed
  - Managed transparently on your behalf
  - It relies on low-level thread fork/join methodology to implement parallelism
    - The developer is spared the details

- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program
OpenMP: Library Support

- Runtime environment routines:
  - Modify/check the number of threads
    
    \[
    \text{omp\_}[\text{set}|\text{get}]\text\_num\_threads() \\
    \text{omp\_get\_thread\_num()} \\
    \text{omp\_get\_max\_threads()} \\
    \]
  
  - Are we in a parallel region?
    
    \[
    \text{omp\_in\_parallel()} \\
    \]
  
  - How many processors in the system?
    
    \[
    \text{omp\_get\_num\_procs()} \\
    \]
  
  - Explicit locks
    
    \[
    \text{omp\_[set|unset]\_lock()} \\
    \]
  
- Many more...

https://computing.llnl.gov/tutorials/openMP/
A Few Syntax Details to Get Started

- Picking up the API - header file in C, or Fortran 90 module
  
  ```
  #include "omp.h"
  use omp_lib
  ```

- Most OpenMP constructs are compiler directives or pragmas

  - For C and C++, the pragmas take the form:
    ```
    #pragma omp construct [clause [clause]...]
    ```

  - For Fortran, the directives take one of the forms:
    ```
    C$OMP construct [clause [clause]...] 
    !$OMP construct [clause [clause]...] 
    !$OMP construct [clause [clause]...] 
    ```
Why Compiler Directive and/or Pragmas?

- One of OpenMP's design principles: the same code, with no modifications, can run either on a one core machine or a multiple core machine.

- Therefore, you have to “hide” all the compiler directives behind Comments and/or Pragmas.

- These directives picked up by the compiler only if you instruct it to compile in OpenMP mode.
  - Example: Visual Studio – you have to have the /openmp flag on in order to compile OpenMP code.
  - Also need to indicate that you want to use the OpenMP API by having the right header included: #include <omp.h>
OpenMP, Compiling Using the Command Line

- Method depends on compiler

- **GCC:**
  
  ```
  $ g++ -o integrate_omp integrate_omp.c -fopenmp
  ```

- **ICC:**
  
  ```
  $ icc -o integrate_omp integrate_omp.c -openmp
  ```

- **MSVC (not in the express edition):**
  
  ```
  $ cl /openmp integrate_omp.c
  ```
OpenMP Odds and Ends...

- Controlling the number of threads
  - The default number of threads that a program uses when it runs is the number of processors on the machine

- For the C Shell: ```setenv OMP_NUM_THREADS number```

- For the Bash Shell: ```export OMP_NUM_THREADS=number```

- Timing:

```c
#include <omp.h>
stime = omp_get_wtime();
mylongfunction();
etime = omp_get_wtime();
total=etime-stime;
```
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization

- Advanced topics
Parallel Region & Structured Blocks (C/C++)

- Most OpenMP constructs apply to **structured blocks**
  - **structured block**, definition: a block with one point of entry at the top and one point of exit at the bottom
  - The only “branches” allowed are exit() function calls

### A structured block

```c
#pragma omp parallel
{
  int id = omp_get_thread_num();
  more: res[id] = do_big_job(id);
  if ( not_conv(res[id]) ) goto more;
}
printf ("All done\n");
```

### Not a structured block

```c
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
  more: res[id] = do_big_job(id);
  if ( conv (res[id]) ) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

There is an implicit barrier at the right “}” curly brace and that’s the point at which the other worker threads complete execution and either go to sleep or spin or otherwise idle.
```c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        int myId = omp_get_thread_num();
        int nThreads = omp_get_num_threads();

        printf("Hello World. I'm thread %d out of %d.\n", myId, nThreads);
        for (int i=0; i<2; i++)
            printf("Iter:%d\n", i);
    }
    printf("All done here...\n");
}
```

Example: Hello World

Here’s my laptop:
Intel Core i5-3210M @ 2.50GHz 3 MB L3 Cache, TDP 35 Watts, Two-Core Four-Thread Processors
OpenMP: Important Remark

- One of the key tenets of OpenMP is that of data independence across parallel jobs

- Specifically, when distributing work among parallel threads it is assumed that there is no data dependency

- Since you place the `omp parallel` directive around some code, it is your responsibility to make sure that data dependency is ruled out
  - Compilers are not smart enough and sometimes and they can’t identify data dependency between what might look as independent parallel jobs
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
    - Data environment
    - Synchronization
- Advanced topics
Work Sharing

- **Work sharing** is the general term used in OpenMP to describe distribution of work across threads.

- Three primary avenues for work sharing in OpenMP:
  - “omp for” construct
  - “omp sections” construct
  - “omp task” construct

Each of them automatically divides work among threads.
“omp for” construct

```c
// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1; i <= N; i++)
        c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

[example above assumes three threads are in the thread team]
Combining Constructs

- These two code segments are equivalent

```c
#pragma omp parallel
{
    #pragma omp for
    for (int i=0; i< MAX; i++) {
        res[i] = huge();
    }
}
```

```c
#pragma omp parallel for
for (int i=0; i< MAX; i++) {
    res[i] = huge();
}
```
The Private Clause

- Reproduces the variable for each task
  - By declaring a variable as being private it means that each thread will have a private copy of that variable
    - The value that Thread-1 stores in x is different than value that Thread-2 stores in variable x
  - Variables are un-initialized; C++ object is default constructed

```c
void* work(float* c, int N) {
    float x, y;
    int i;
    #pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
    }
}
```
The *schedule* Clause

- The *schedule* clause affects how loop iterations are mapped onto threads

```c
schedule(static [,chunk])
```
- Blocks of iterations of size “chunk” assigned to each thread
- Round robin distribution
- Low overhead, may cause load imbalance

```c
schedule(dynamic[,chunk])
```
- Threads grab “chunk” iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

```c
schedule(guided[,chunk])
```
- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than “chunk”
schedule Clause Example

```c
#pragma omp parallel for schedule (static, 8)  
for (int i = start; i <= end; i += 2 )  
{  
    if ( TestForPrime(i) ) gPrimesFound++;  
}
```

- Iterations are divided into chunks of 8
- If start = 3, then first chunk is

\[ i = \{3, 5, 7, 9, 11, 13, 15, 17\} \]
Example, Static Scheduling

- What is the default value of the chunk size?
  - Depends on the underlying hardware
  - Can figure out writing a short code sample:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel for schedule (static)
    for (int i = 0; i<=14; i++) {
        printf("I'm thread %d working on iteration %d\n",omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}
```
Work Plan

- What is OpenMP?
  Parallel regions
  Work sharing – Parallel Sections
  Data environment
  Synchronization

- Advanced topics
Function Level Parallelism

```c
a = alice();
b = bob();
s = boss(a, b);
k = kate();
printf("%6.2f\n", bigboss(s,k));
```

alice, bob, and kate can be computed in parallel
omp sections

- `#pragma omp sections`
- Must be inside a parallel region
- Precedes a code block containing $N$ sub-blocks of code that may be executed concurrently by $N$ threads
- Encompasses each `omp section`, see below

- `#pragma omp section`
- Precedes each sub-block of code within the encompassing block described above
- Enclosed program segments are distributed for parallel execution among available threads
Functional Level Parallelism Using omp sections

```c
#pragma omp parallel sections
{
    #pragma omp section
        double a = alice();
    #pragma omp section
        double b = bob();
    #pragma omp section
        double k = kate();
}

double s = boss(a, b);
printf("%6.2f\n", bigboss(s,k));
```
Advantage of Parallel Sections

- Independent sections of code can execute concurrently → reduces execution time

```c
#pragma omp parallel sections
{
#pragma omp section
    phase1();
#pragma omp section
    phase2();
#pragma omp section
    phase3();
}
```

The pink and green tasks are executed at no additional time-penalty in the shadow of the blue task
#include <stdio.h>
#include <omp.h>

int main()
{
    printf("Start with 2 procs only...

");
    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while ((omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while ((omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while ((omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
}
sections, Example: 2 threads
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 4 procs\n");
    #pragma omp parallel sections num_threads(4)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0); 
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 6.0); 
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0); 
            printf("Finish work 3\n");
        }
    }
    return 0;
}
sections, Example: 4 threads

Start with 4 procs
Start work 1
Start work 2
Start work 3
Finish work 1
Finish work 3
Finish work 2
Press any key to continue . . .
Work Plan

- What is OpenMP?
  Parallel regions
  Work sharing – Tasks
  Data environment
  Synchronization

- Advanced topics
OpenMP Tasks

- **Task** – Most important feature added as of OpenMP 3.0 version

- Allows parallelization of irregular problems
  - Unbounded loops (not clear how many iterations – see next example)
  - Recursive algorithms
  - Producer/consumer

- Start next with a motivating example
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel for schedule (static)
    for (int i = 0; i <= 14; i++) {
        printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}
#include <stdio.h>
#include <omp.h>

int getUpperBound(int i, int N){
    if (i <= N)
        return N;
    else
        return 0;
}

int main() {
    int upperB = 14;

    for (int i = 0; i <= getUpperBound(i,upperB); i++){
        printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}
```c
#include <stdio.h>
#include <omp.h>

int getUpperBound(int i, int N){
    if (i <= N)
        return N;
    else
        return 0;
}

int main() {
    int upperB = 14;

    #pragma omp parallel for schedule (static)
    for (int i = 0; i <= getUpperBound(i,upperB); i++){
        printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}
```
Tasks: What Are They?

- Tasks are independent units of work
- A thread is assigned to perform a task
- Tasks might be executed immediately or might be deferred
  - The OS & runtime decide which of the above
- Tasks are composed of
  - code to execute
  - data environment
  - internal control variables (ICV)
Tasks: What Are They?

[More specifics…]

- **Code to execute**
  - The literal code in your program enclosed by the task directive

- **Data environment**
  - The shared & private data manipulated by the task

- **Internal control variables**
  - Thread scheduling and environment variables

- More formal definition: A task is a specific instance of executable code and its data environment, generated when a thread encounters a task construct

- Two activities: (1) packaging, and (2) execution
  - A thread packages new instances of a task (code and data)
  - Some thread in the team executes the task at some later time
using namespace std;
typedef list<double> LISTDBL;

void doSomething(LISTDBL::iterator& itrtr) {
    *itrtr *= 2.;
}

int main() {
    LISTDBL test; // default constructor
    LISTDBL::iterator it;

    for( int i=0;i<4;++i)
        for( int j=0;j<8;++j) test.insert(test.end(), pow(10.0,i+1)+j);
    for( it = test.begin(); it!= test.end(); it++ ) cout << *it << endl;

    it = test.begin();
    #pragma omp parallel num_threads(8)
    {
        #pragma omp single
        {
            while( it != test.end() ) {
                #pragma omp task private(it)
                {
                    doSomething(it);
                }
                it++;
            }
        }
        for( it = test.begin(); it != test.end(); it++ ) cout << *it << endl;
    }
    return 0;
}
Compile like:

```
$ g++ -o testOMP.exe testOMP.cpp
```
Task Construct – Explicit Task View

- A team of threads is created at the `omp parallel` construct
- A single thread is chosen to execute the while loop – let’s call this thread “L”
- Thread L runs the while loop, creates tasks, and fetches next pointers
- Each time L crosses the `omp task` construct it generates a new task and has a thread assigned to it
- Each task run by one thread
- All tasks complete at the barrier at the end of the parallel region’s construct
- Each task has its own stack space that will be destroyed when the task is completed
  - See example in a little bit

```c
#pragma omp parallel
//threads are ready to go now
{
    #pragma omp single
    { // block 1
        node *p = head_of_list;
        while (p!=listEnd) { //block 2
            #pragma omp task private(p)
            process(p);
            p = p->next; //block 3
        }
    }
}
```
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
//threads are ready to go now
{
    #pragma omp single
    {
        // block 1
        node *p = head_of_list;
        while (p) {
            #pragma omp task private(p)
            {
                process(p);
            }
            p = p->next;  //block 3
        }
    }
}

[IOMPP]→

How about synchronization issues?
Tasks: Synchronization Issues

- Setup:
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guarantee completion of Task A before invoking the execution of Task B

- Tasks are guaranteed to be complete at thread or task barriers:
  - At the directive: `#pragma omp barrier`
  - At the directive: `#pragma omp taskwait`
Task Completion Example

```c
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

- Multiple foo tasks created here - one for each thread
- All foo tasks guaranteed to be completed here
- One bar task created here
- bar task guaranteed to be completed here
Comments: sections vs. tasks

- **sections** have a “static” attribute: things are mostly settled at compile time

- The **tasks** construct is more recent and more sophisticated
  - They have a “dynamic” attribute: things are figured out at run time and the construct counts under the hood on the presence of a scheduling agent
  - They can encapsulate any block of code
    - Can handle nested loops and scenarios when the number of jobs is not clear
  - The runtime generates and executes the tasks, either at implicit synchronization points in the program or under explicit control of the programmer

- **NOTE:** It’s the developer responsibility to ensure that different tasks can be executed concurrently; i.e., there is no data dependency
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data scoping
  - Synchronization

- Advanced topics
Data Scoping – What’s shared

- OpenMP uses a shared-memory programming model

- **Shared variable** - a variable that can be read or written by multiple threads

- **shared** clause can be used to make items explicitly shared
  - Global variables are shared by default among tasks
  - Other examples of variables being shared among threads
    - File scope variables
    - Namespace scope variables
    - Variables with const-qualified type having no mutable member
    - Static variables which are declared in a scope inside the construct
Data Scoping – What’s Private

- Not everything is shared...
  - Examples of implicitly determined PRIVATE variables:
    - Stack (local) variables in functions called from parallel regions
    - Automatic variables within a statement block
    - Loop iteration variables
    - Implicitly declared private variables within tasks will be treated as firstprivate

- firstprivate
  - Specifies that each thread should have its own instance of a variable
  - Data is initialized using the value of the variable using the same name from the master thread
Example: 
private vs. firstprivate

```c
#include <stdio.h>
#include <omp.h>

int main(void) {
    int i = 10;

#pragma omp parallel private(i)
    {
        int threadID = omp_get_thread_num();
        printf("thread %d: i = %d\n", threadID, i);
        i = 1000 + threadID;
    }

    printf("i = %d\n", i);
    return 0;
}
```

[stackoverflow]→
Example:
private vs. firstprivate

```c
#include <stdio.h>
#include <omp.h>

int main(void) {
    int i = 10;

    #pragma omp parallel firstprivate(i)
    {
        int threadID =omp_get_thread_num();
        printf("threadID + i = %d\n", threadID+i);
    }

    printf("i = %d\n", i);

    return 0;
}
```

[stackoverflow]→
Other Tidbits

- There is a *lastprivate* flavor of private variable
  - The enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration of the work-sharing construct (*for*, *section*, *task*)
Data Scoping – The Basic Rule

- When in doubt, explicitly indicate who’s what
  - Data scoping: one of the most common sources of errors in OpenMP
```c
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
    tid =omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }

    printf("Thread %d starting...\n",tid);

#pragma omp sections nowait
{
    #pragma omp section
    {
        printf("Thread %d doing section 1\n",tid);
        for (i=0; i<N; i++)
        {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
    }

    #pragma omp section
    {
        printf("Thread %d doing section 2\n",tid);
        for (i=0; i<N; i++)
        {
            d[i] = a[i] * b[i];
            printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
        }
    }

    /* end of sections */

    printf("Thread %d done.\n",tid);
} /* end of parallel section */
```
A, index, and count are shared by all threads, but temp is local to each thread.
#include <stdio.h>
#include <omp.h>

int fib(int);

int main()
{
    int n = 10;
    omp_set_num_threads(4);

    #pragma omp parallel
    {
        #pragma omp single
        printf("fib(%d) = %d\n", n, fib(n));
    }
}
Example: Data Scoping Issue - `fib`

Assume that the parallel region exists outside of `fib` and that `fib` and the tasks inside it are in the dynamic extent of a parallel region.

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

- **n** is private in both tasks
- **x** is a private variable
- **y** is a private variable
- This is very important here
- What’s wrong here?

Values of the private variables not available outside of tasks
Example: Data Scoping Issue - \texttt{fib}

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task
    {
        x = fib(n-1);
    }
    #pragma omp task
    {
        y = fib(n-2);
    }
    #pragma omp taskwait
    return x+y
}
```

Values of the private variables not available outside of tasks

x is a private variable
y is a private variable
Example: Data Scoping Issue - \texttt{fib}

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait

    return x+y;
}
```

- \texttt{n} is private in both tasks
- \texttt{x} & \texttt{y} are now shared
- We need both values to compute the sum

The values of the \texttt{x} & \texttt{y} variables will be available outside each task construct – after the taskwait
#include <stdio.h>
#include <omp.h>

int main(void) {
    const int N = 3;
    int a[3] = { 2, 4, 6 };   // Array a
    int b[3] = { 1, 3, 5 };   // Array b
    int c[3], d[3];
    int i, tid, nthreads;

    #pragma omp parallel private(i,tid) shared(a,b)
    {
        tid = omp_get_thread_num();
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

        printf("Thread %d starting...\n", tid);
    }

    #pragma omp sections
    {
        #pragma omp section
        {
            printf("Thread %d doing section 1\n", tid);
            for (i = 0; i < N; i++) {
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %d\n", tid, i, c[i]);
            }
        }

        #pragma omp section
        {
            printf("Thread %d doing section 2\n", tid);
            for (i = 0; i < N; i++) {
                d[i] = a[i] * b[i];
                printf("Thread %d: d[%d]= %d\n", tid, i, d[i]);
            }
        }
    } /* end of sections */

    printf("Thread %d done.\n", tid);
} /* end of parallel section */
for (i = 0; i < N; i++) {
    printf("c[%d] = %d AND d[%d]= %d\n", i, c[i], i, d[i]);
}

    return 0;
}
```c
#include <stdio.h>
#include <omp.h>

int main(void) {
    const int N = 3;
    int a[3] = { 2, 4, 6};
    int b[3] = { 1, 3, 5};
    int c[3], d[3];
    int i, tid, nthreads;

    #pragma omp parallel private(i,tid, c, d) shared(a,b)
    {
        tid = omp_get_thread_num();
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

        printf("Thread %d starting...\n", tid);

        #pragma omp sections
        {
            #pragma omp section
            {
                printf("Thread %d doing section 1\n", tid);
                for (i = 0; i < N; i++) {
                    c[i] = a[i] + b[i];
                    printf("Thread %d: c[%d]= %d\n", tid, i, c[i]);
                }
            }

            #pragma omp section
            {
                printf("Thread %d doing section 2\n", tid);
                for (i = 0; i < N; i++) {
                    d[i] = a[i] * b[i];
                    printf("Thread %d: d[%d]= %d\n", tid, i, d[i]);
                }
            }
        }
        /* end of sections */

        printf("Thread %d done.\n", tid);
    } /* end of parallel section */

    for (i = 0; i < N; i++) {
        printf("c[%d] = %d AND d[%d]= %d\n", i, c[i], i, d[i]);
    }

    return 0;
}
```

Work Plan

What is OpenMP?
- Parallel regions
- Work sharing
- Data environment
- Synchronization
- Advanced topics
Implicit Barriers

- Several OpenMP constructs have *implicit* barriers
  - *parallel* – necessary barrier – cannot be removed
  - *for*
  - *single*

- Unnecessary barriers hurt performance and can be removed with the *nowait* clause
  - The *nowait* clause is applicable to:
    - *for* clause
    - *single* clause
Nowait Clause

- Use when threads unnecessarily wait between independent computations

```
#pragma omp for nowait
for(...)
{
...
};
```

```
#pragma single nowait
{
...
}
```

```
#pragma omp for schedule(dynamic,1) nowait
for(int i=0; i<n; i++)
    a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
for(int j=0; j<m; j++)
    b[j] = bigFunc2(j);
```

Credit: IOMPP
Barrier Construct

- Explicit barrier synchronization
- Each thread waits until all threads arrive

```
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A,B); //input is A, output is B
    #pragma omp barrier
    DoSomeWork(B,C); //input is B, output is C
}
```

Credit: IOMPP
Atomic Construct

- Applies only to simple update of memory location
- Special case of a critical section, to be discussed shortly
  - Atomic introduces less overhead than critical

```c
#pragma omp parallel for shared(x, y, index)
  for (i = 0; i < n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
    y[i] += work2(i);
  }
```

Credit: IOMPP
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
Race Condition

- Definition, *race condition*: two or more threads access a shared variable at the same time.
  - Leads to nondeterministic behavior

- For example, suppose that *area* is shared and both Thread A and Thread B are executing the statement

  \[ \text{area } += \ 4.0 \ / \ (1.0 \ + \ x^2) ; \]
Two Possible Scenarios

Order of thread execution causes non determinent behavior in a data race

Credit: IOMPP
Protect Shared Data

- The **critical** construct: protects access to shared, modifiable data
- The critical section allows only one thread to enter it at a given time

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

Credit: IOMPP
OpenMP Critical Construct

```
#define omp critical [(lock_name)]
```

- Defines a critical region on a structured block

Threads wait their turn – only one at a time calls `consum()` thereby protecting `RES` from race conditions.

Naming the critical construct `RES_lock` is optional but highly recommended.

```c
float RES;
#pragma omp parallel
{
#pragma omp for
    for(int i=0; i<niters; i++) {
        float B = big_job(i);
        #pragma omp critical (RES_lock)
        consum(B, RES);
    }
}
```

The “for” loop includes material from IOMPP

Includes material from IOMPP
reduction Example

```c
#pragma omp parallel for reduction(+:sum)
    for(i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
```

- Local copy of `sum` for each thread engaged in the reduction is private
  - Each local sum initialized to the identity operand associated with the operator that comes into play
    - Here we have “+”, so it’s a zero (0)

- All local copies of `sum` added together and stored in “global” variable

Credit: IOMPP
OpenMP reduction Clause

- The variables in `list` will be shared in the enclosing parallel region

- Here’s what happens inside the parallel or work-sharing construct:
  - A private copy of each list variable is created and initialized depending on the “op”
  - These copies are updated locally by threads

- At end of construct, local copies are combined through “op” into a single value
OpenMP Reduction Example: Numerical Integration

\[ \int_{0}^{1} \frac{4.0}{(1 + x^2)} \, dx = \pi \]

```
static long num_steps = 100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n", pi);
}
```
OpenMP Reduction Example: Numerical Integration

```c
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

int main(int argc, char* argv[]) {
    int num_steps = atoi(argv[1]);
    double step = 1./(double(num_steps));
    double sum;

    #pragma omp parallel for reduction(+:sum)
    {
        for(int i=0; i<num_steps; i++) {
            double x = (i + .5)*step;
            sum += 4.0/(1.+ x*x);
        }
    }

double my_pi = sum*step;
printf("Value of integral is: \%f\n", my_pi);

    return 0;
}
```
OpenMP Reduction Example:
Output

[negrut@euler24 CodeBits]$ g++ testOMP.cpp -o test.exe
[negrut@euler24 CodeBits]$ ./test.exe 100000
Value of integral is: 3.141593
C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Credit: IOMPP
Example: Variable Scoping Aspects

- Consider parallelizing the following code

```c
int main() {
    const int n=20;
    int a[n];
    for( int i=0; i<n; i++ )
        a[i] = i;

    //this is the part that needs to
    //be parallelized
    caller(a, n);

    for( int i=0; i<n; i++ )
        printf("a[%d]=%d\n", i, a[i]);

    return 0;
}

void caller(int *a, int n) {
    int i, j, m=3;
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}

void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<n; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}
```
Program Output

- Looks good
  - The value of the counter increases each time you hit the “callee” subroutine

- If you run the executable 20 times, you get the same results 20 times
First Attempt to Parallelize

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Scope</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>n</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>i</td>
<td>private</td>
<td>Parallel loop index</td>
</tr>
<tr>
<td>j</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>m</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>k</td>
<td>private</td>
<td>Automatic variable/parallel region</td>
</tr>
<tr>
<td>x</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*x</td>
<td>shared</td>
<td>(actually a)</td>
</tr>
<tr>
<td>y</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*y</td>
<td>private</td>
<td>(actually k)</td>
</tr>
<tr>
<td>z</td>
<td>private</td>
<td>(actually j)</td>
</tr>
<tr>
<td>ii</td>
<td>private</td>
<td>Local stack variable in called function</td>
</tr>
<tr>
<td>cv</td>
<td>shared</td>
<td>Declared static (like global)</td>
</tr>
</tbody>
</table>
Program Output, First Attempt to Parallelize

- Looks bad...
  - The values in array “a” are all over the map
  - The value of the counter “cv” changes chaotically within “callee”
  - The function “callee” gets hit a random number of times (should be hit 100 times). Example:
    ```bash
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 70
    ```

- If you run executable 20 times, you get different results

- One of the problems is that “j” is shared
Second Attempt to Parallelize

- Declare the inner loop variable “j” as a private variable within the parallel loop

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for private(j)
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output, Second Attempt to Parallelize

- Looks better
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 100

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely (it’s shared)
  - Fortunately, it’s not used in this code for any subsequent computation

- Q: How would you fix this issue with the static variable?
  - Not necessarily to print the values in increasing order, but to make sure there are no race conditions
Slightly Better Solution…

- Declare the inner loop index “j” only inside the parallel segment
  - After all, it’s only used there
  - You get rid of the “private” attribute, less constraints on the code, increasing the opportunity for code optimization at compile time

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (int j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

Used here, then you should declare here (common sense…)

Program Output, Parallelized Code

- It looks good
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    ```
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 100
    ```

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely
  - No reason for this behavior to change
Concluding Remarks on the OpenMP API
OpenMP: 30,000 Feet Perspective

- Good momentum behind OpenMP owing to the ubiquity of the multi-core chips
- Shared memory, thread-based parallelism
- Relies on the programmer defining parallel regions
- Fork/join model

- Industry-standard shared memory programming model
  - First version released in 1997
  - OpenMP 4.0 – complete specifications released in July 2013

[Rebecca Hartman-Baker]→
OpenMP
The 30,000 Feet Perspective

- Nomenclature:
  - Multicore Communication API (MCAPI)
  - Multicore Resource-sharing API (MRAPI)
  - Multicore Task Management API (MTAPI)
The OpenMP API

- The OpenMP API is a combination of
  - Directives
    - Example: \#pragma omp task
  - Runtime library routines
    - Example: int omp_get_thread_num(void)
  - Environment variables
    - Example: setenv OMP_SCHEDULE "guided, 4"
The “directives” fall into three categories

- Expression of parallelism (flow control)
  - Example: `#pragma omp parallel for`

- Data sharing among threads (communication)
  - Example: `#pragma omp parallel for private(x,y)`

- Synchronization (coordination or interaction)
  - Example: `#pragma omp barrier`
OpenMP 4.0: Subset of Run-Time Library OpenMP Routines

1. `omp_set_num_threads`
2. `omp_get_num_threads`
3. `omp_get_max_threads`
4. `omp_get_thread_num`
5. `omp_get_thread_limit`
6. `omp_get_num_procs`
7. `omp_in_parallel`
8. `omp_set_dynamic`
9. `omp_get_dynamic`
10. `omp_set_nested`
11. `omp_get_nested`
12. `omp_set_schedule`
13. `omp_get_schedule`
14. `omp_set_max_active_levels`
15. `omp_get_max_active_levels`
16. `omp_get_level`
17. `omp_get_ancestor_thread_num`
18. `omp_get_team_size`
19. `omp_get_active_level`
20. `omp_init_lock`
21. `omp_destroy_lock`
22. `omp_set_lock`
23. `omp_unset_lock`
24. `omp_test_lock`
25. `omp_init_nest_lock`
26. `omp_destroy_nest_lock`
27. `omp_set_nest_lock`
28. `omp_unset_nest_lock`
29. `omp_test_nest_lock`
30. `omp_get_wtime`
31. `omp_get_wtick`
OpenMP: Environment Variables

- **OMP_SCHEDULE**
  - Example: `setenv OMP_SCHEDULE "guided, 4"

- **OMP_NUM_THREADS**
  - Sets the maximum number of threads to use during execution.
  - Example: `setenv OMP_NUM_THREADS 8`

- **OMP_DYNAMIC**
  - Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE
  - Example: `setenv OMP_DYNAMIC TRUE`

- **OMP_NESTED**
  - Enables or disables nested parallelism. Valid values are TRUE or FALSE
  - Example: `setenv OMP_NESTED TRUE`
OpenMP: Environment Variables
[select env variables]

- **OMP_STACKSIZE**
  - Controls the size of the stack for created (non-Master) threads.

- **OMP_WAIT_POLICY**
  - Provides a hint to an OpenMP implementation about the desired behavior of waiting threads.

- **OMP_MAX_ACTIVE_LEVELS**
  - Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. Example:
    - `setenv OMP_MAX_ACTIVE_LEVELS 2`

- **OMP_THREAD_LIMIT**
  - Sets the number of OpenMP threads to use for the whole OpenMP program
  - Example:
    - `setenv OMP_THREAD_LIMIT 8`
Attractive Features of OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)

- Can implement complex algorithms

- Code size grows only modestly

- Expression of parallelism flows clearly, code is easy to read

- Single source code for OpenMP and non-OpenMP
  - Non-OpenMP compilers simply ignore OMP directives
OpenMP, Some Caveats

- There is a lag between the moment a new specification is released and the time a compiler is capable of handling all of its aspects
  - Intel’s compiler is probably most up to speed

- OpenMP threads are heavy
  - Good for handling parallel tasks
  - Not so good at handling fine large scale grain parallelism
Further Reading, OpenMP

- Michael Quinn (2003) Parallel Programming in C with MPI and OpenMP
- LLNL OpenMP Tutorial, [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)
- OpenMP.org, [http://openmp.org/](http://openmp.org/)
- OpenMP 3.0 API Summary Cards:
  - C/C++: [http://openmp.org/mp-documents/OpenMP-4.0-C.pdf](http://openmp.org/mp-documents/OpenMP-4.0-C.pdf)
- [http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf](http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf)