CS 677: Parallel Programming for Many-core Processors
Lecture 12

Instructor: Philippos Mordohai

Webpage: mordohai.github.io

E-mail: Philippos.Mordohai@stevens.edu

Final Project Presentations

- April 29
  - **Submit PPT/PDF file by 5 pm**
  - **Test your microphones, be ready to share your screen and present on zoom**
  - 8 min presentation + 2 min Q&A
- Counts for 15% of total grade

Final Project Presentations

- Target audience: fellow classmates
- Content:
  - Problem description
    - What is the computation and why is it important?
  - Suitability for GPU acceleration
    - Amdahl's Law: describe the inherent parallelism. Argue that it is close to 100% of computation.
    - Compare with CPU version

Final Project Presentations

- Content (cont.):
  - GPU Implementation
    - Which steps of the algorithm were ported to the GPU?
    - Work load allocation to threads
    - Use of resources (registers, shared memory, constant memory, etc.)
    - Occupancy achieved
  - Results
    - Experiments performed
    - Timings and comparisons between CPU and multiple GPU versions

Final Report

- Due May 13 (11:59 pm)
  - Syllabus says May 11, but there is no reason
- 6 pages excluding figures, tables and references
  - Suggested format: single-column, 11-point font and one-inch margins all around
- Content
  - See presentation instructions
  - Do not repeat course material
- Counts for 20% of total grade
- NO LATE SUBMISSIONS

Outline

- OpenCL
  - Image Convolution
- OpenACC
- OpenMP

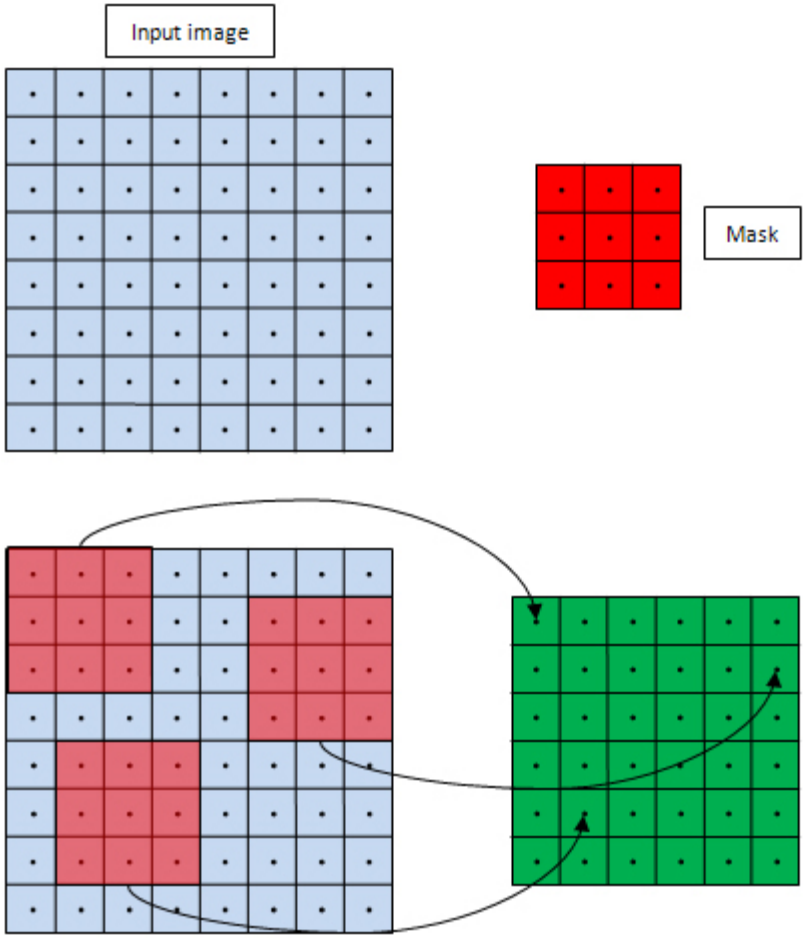
Image Convolution Using OpenCL™

Udeepta Bordoloi,
ATI Stream Application Engineer

10/13/2009

Note: ATI Stream Technology is now called AMD Accelerated Parallel Processing (APP) Technology.

Step 1 - The Algorithm



- Ignore boundaries
- Output size:

(input_image_width – filter_width + 1) by
(input_image_height – filter_width + 1)

C Version

```
void Convolve(float * pInput, float * pFilter, float
   * pOutput, const int nInWidth, const int nWidth,
   const int nHeight,
const int nFilterWidth, const int nNumThreads)
{
   for (int yOut = 0; yOut < nHeight; yOut++)
   {
       const int yInTopLeft = yOut;
       for (int xOut = 0; xOut < nWidth; xOut++)
       {
             const int xInTopLeft = xOut;
             float sum = 0;
```

C Version (2)

```
for (int r = 0; r < nFilterWidth; r++)
{
      const int idxFtmp = r * nFilterWidth;
      const int yIn = yInTopLeft + r;
      const int idxIntmp = yIn * nInWidth +
                  xInTopLeft;
      for (int c = 0; c < nFilterWidth; c++)
      {
            const int idxF = idxFtmp + c;
            const int idxIn = idxIntmp + c;
            sum += pFilter[idxF]*pInput[idxIn];
      }
} //for (int r = 0…
```

C Version (3)

```
            const int idxOut = yOut * nWidth + xOut;
            pOutput[idxOut] = sum;
        } //for (int xOut = 0…
    } //for (int yOut = 0…
}
```

Parameters

```
struct paramStruct
{
   int nWidth; //Output image width
   int nHeight; //Output image height
   int nInWidth; //Input image width
   int nInHeight; //Input image height
   int nFilterWidth; //Filter size is nFilterWidth X
                     //nFilterWidth
   int nIterations; //Run timing loop for nIterations
   //Test CPU performance with 1,4,8 etc. OpenMP threads
   std::vector ompThreads;
   int nOmpRuns; //ompThreads.size()
   bool bCPUTiming; //Time CPU performance
} params;
```

OpenMP for Comparison

```
//This #pragma splits the work between multiple threads
#pragma omp parallel for num_threads(nNumThreads)
for (int yOut = 0; yOut < nHeight; yOut++)
...

void InitParams(int argc, char* argv[])
{
…
// time the OpenMP convolution performance with
// different numbers of threads
   params.ompThreads.push_back(4);
   params.ompThreads.push_back(1);
   params.ompThreads.push_back(8);
   params.nOmpRuns = params.ompThreads.size();
}
```

First Kernel

```
__kernel void Convolve(const __global float * pInput,
  __constant float * pFilter, __global float * pOutput,
  const int nInWidth, const int nFilterWidth)
{
  const int nWidth = get_global_size(0);
  const int xOut = get_global_id(0);
  const int yOut = get_global_id(1);
  const int xInTopLeft = xOut;
  const int yInTopLeft = yOut;

  float sum = 0;
```

First Kernel (2)

```
  for (int r = 0; r < nFilterWidth; r++)
  {
      const int idxFtmp = r * nFilterWidth;
      const int yIn = yInTopLeft + r;
      const int idxIntmp = yIn * nInWidth + xInTopLeft;

      for (int c = 0; c < nFilterWidth; c++)
      {
            const int idxF = idxFtmp + c;
            const int idxIn = idxIntmp + c;
            sum += pFilter[idxF]*pInput[idxIn];
      }
  } //for (int r = 0…
  const int idxOut = yOut * nWidth + xOut;
  Output[idxOut] = sum;
}
```

Initialize OpenCL

```
cl_context context =
  clCreateContextFromType(…,CL_DEVICE_TYPE_CPU,…);

// get list of devices - quad core counts as one device
size_t listSize;
/* First, get the size of device list */
clGetContextInfo(context, CL_CONTEXT_DEVICES, …,
  &listSize);
/* Now, allocate the device list */
cl_device_id devices = (cl_device_id *)malloc(listSize);
/* Next, get the device list data */
clGetContextInfo(context, CL_CONTEXT_DEVICES, listSize,
  devices, …);
```

Initialize OpenCL (2)

```
cl_command_queue queue = clCreateCommandQueue(context,
  devices[0], …);

cl_program program = clCreateProgramWithSource(context,
  1, &source, …);

clBuildProgram(program, 1, devices, …);

cl_kernel kernel = clCreateKernel(program, "Convolve",
  …);

// get error messages
clGetProgramBuildInfo(program, devices[0],
  CL_PROGRAM_BUILD_LOG, …);
```

Initialize Buffers

```
cl_mem inputCL = clCreateBuffer(context,
   CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
   host_buffer_size, host_buffer_ptr, …);

//If the device is a GPU (CL_DEVICE_TYPE_GPU), we can
// explicitly copy data to the input image buffer on the
// device:
clEnqueueWriteBuffer(queue, inputCL, …, host_buffer_ptr,
      …);

// And copy back from the output image buffer after the
// convolution kernel execution.
clEnqueueReadBuffer(queue, outputCL, …, host_buffer_ptr,
      …);
```

Execute Kernel

```
/* input buffer, arg 0 */
clSetKernelArg(kernel, 0, sizeof(cl_mem),
      (void *)&inputCL);
/* filter buffer, arg 1 */
clSetKernelArg(kernel, 1, sizeof(cl_mem),
      (void *)&filterCL);
/* output buffer, arg 2 */
clSetKernelArg(kernel, 2, sizeof(cl_mem),
      (void *)&outputCL);
/* input image width, arg 3*/
clSetKernelArg(kernel, 3, sizeof(int),
      (void *)&nInWidth);
/* filter width, arg 4*/
clSetKernelArg(kernel, 4, sizeof(int),
      (void *)&nFilterWidth);
```

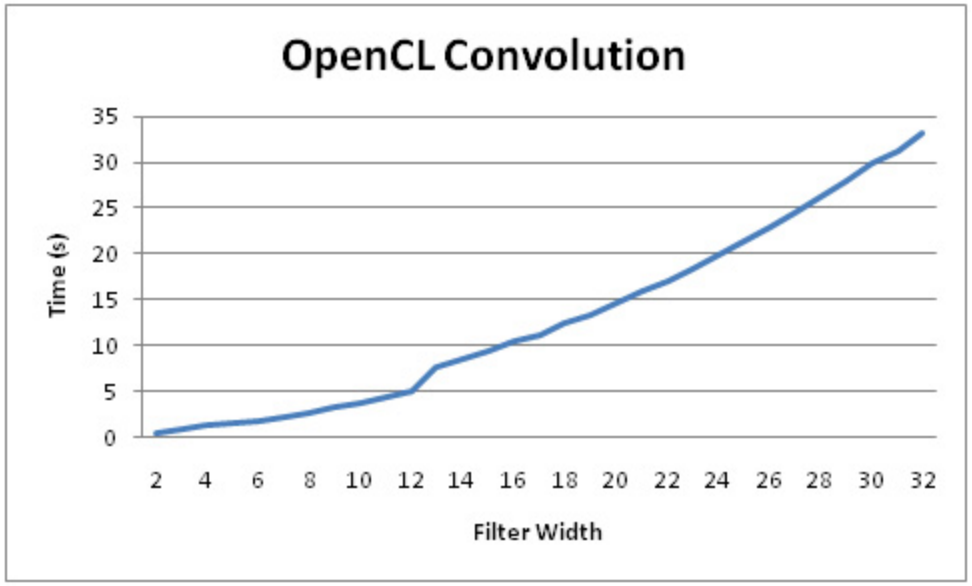
Execute Kernel

```
clEnqueueNDRangeKernel(queue, kernel,
      data_dimensionality, …, total_work_size,
      work_group_size, …);

// release all buffers
clReleaseBuffer(inputCL);
...

// release all resources
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
```

Timing

```
clFinish(queue); //Timer Started here();
for (int i = 0; i < nIterations; i++)
      clEnqueueNDRangeKernel(…);
clFinish(queue); //Timer Stopped here();
//Average Time = ElapsedTime()/nIterations;
```

clFinish() call before both starting and stopping the timer ensures that we time the kernel execution activity to its completion and nothing else



On 4-core AMD Phenom treated as a single device by OpenCL

C++ Bindings

```
cl_context context =
        clCreateContextFromType(…,CL_DEVICE_TYPE_CPU,…);
cl::Context context = cl::Context(CL_DEVICE_TYPE_CPU);

// get list of devices - quad core counts as one device
size_t listSize;
/* First, get the size of device list */
clGetContextInfo(context, CL_CONTEXT_DEVICES, …, &listSize);
/* Now, allocate the device list */
cl_device_id devices = (cl_device_id *)malloc(listSize);
/* Next, get the device list data */
clGetContextInfo(context, CL_CONTEXT_DEVICES, listSize,
        devices, …);
std::vector<cl::Device> devices = context.getInfo();
```

See https://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf

C++ Bindings (2)

```
cl::CommandQueue queue = cl::CommandQueue(context, devices[0]);

cl::Program program = cl::Program(context, …);

program.build(devices);

cl::Kernel kernel = cl::Kernel(program, "Convolve");

string str = program.getBuildInfo(devices[0]);

// Buffer init is similar to C version
// using methods of queue
```

Execute Kernel

```
 /* input buffer, arg 0 */
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&inputCL);
kernel.setArg(0, inputCL);

/* filter buffer, arg 1 */
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&filterCL);
kernel.setArg(1, filterCL);

// etc.


queue.clEnqueueNDRangeKernel(kernel, …, total_work_size,
       work_group_size, …);
```

Loop Unrolling

```
__kernel void Convolve_Unroll(const __global float * pInput,
      __constant float * pFilter, __global float * pOutput,
      const int nInWidth, const int nFilterWidth)
{
      const int nWidth = get_global_size(0);
      const int xOut = get_global_id(0);
      const int yOut = get_global_id(1);
      const int xInTopLeft = xOut;
      const int yInTopLeft = yOut;

      float sum = 0;
      for (int r = 0; r < nFilterWidth; r++)
      {
            const int idxFtmp = r * nFilterWidth;
            const int yIn = yInTopLeft + r;
            const int idxIntmp = yIn * nInWidth + xInTopLeft;
```

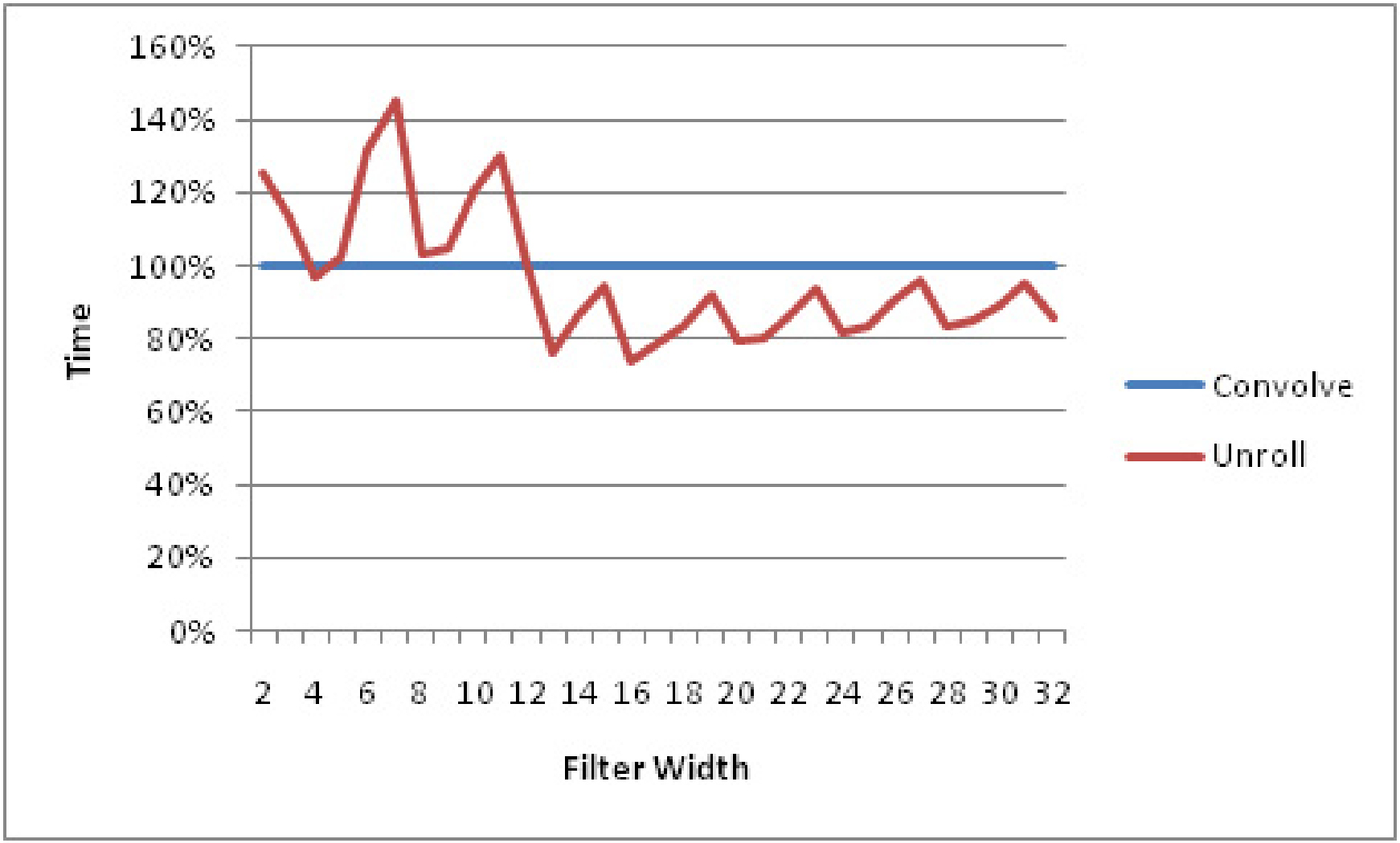
Loop Unrolling (2)

```
int c = 0;
while (c <= nFilterWidth-4)
{
        int idxF = idxFtmp + c;
        int idxIn = idxIntmp + c;
        sum += pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        sum += pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        sum += pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        sum += pFilter[idxF]*pInput[idxIn];
        c += 4;
}
```

Loop Unrolling (3)

```
                for (int c1 = c; c1 < nFilterWidth; c1++)
                {
                        const int idxF = idxFtmp + c1;
                        const int idxIn = idxIntmp + c1;
                        sum += pFilter[idxF]*pInput[idxIn];
                }
        } //for (int r = 0…
        const int idxOut = yOut * nWidth + xOut;
        pOutput[idxOut] = sum;
}

// what does this do?
```

Performance



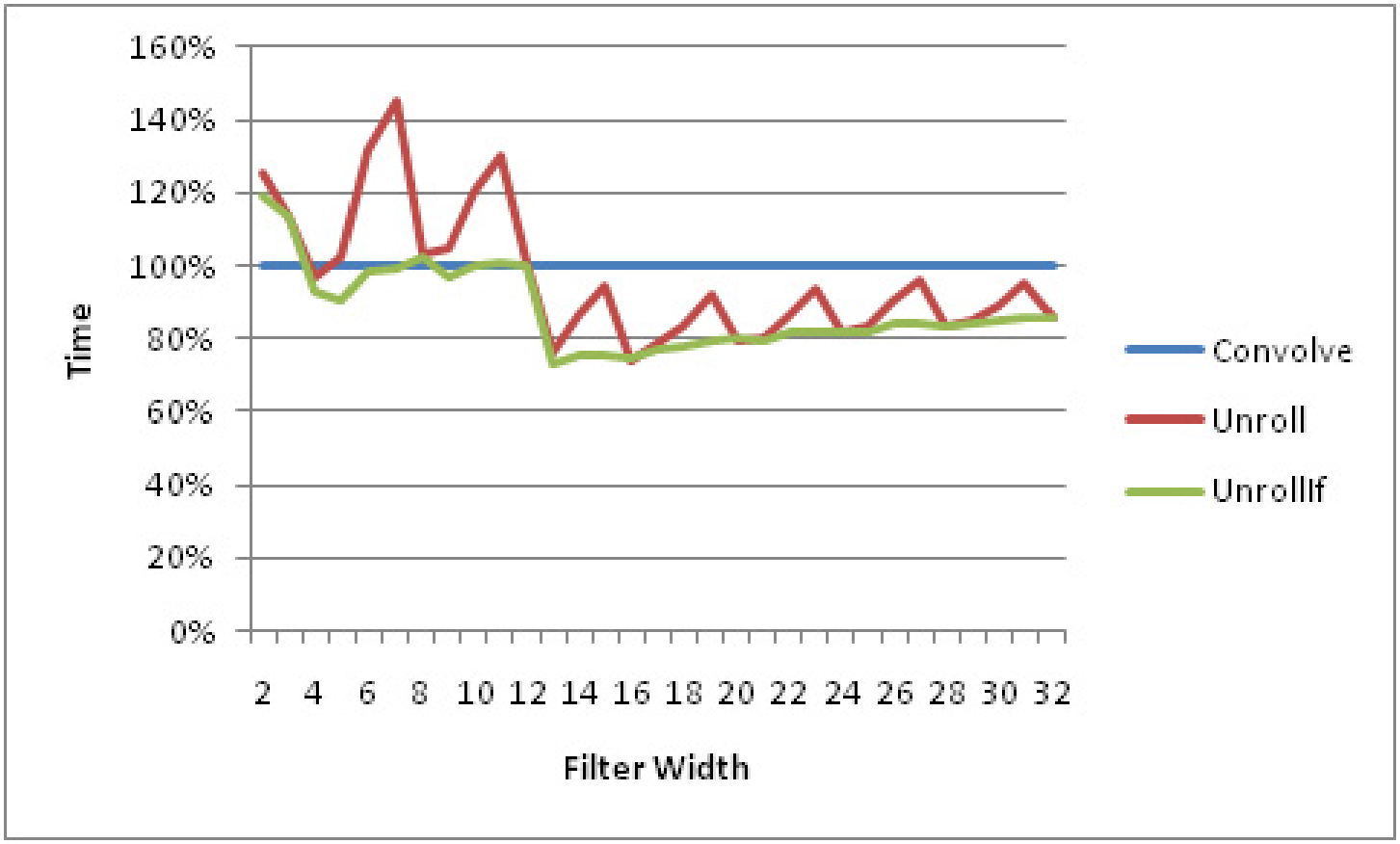
Unrolled Kernel 2 (if Kernel)

```
// last loop
int cMod = nFilterWidth - c;
if (cMod == 1)
{
        int idxF = idxFtmp + c;
        int idxIn = idxIntmp + c;
        sum += pFilter[idxF]*pInput[idxIn];
}
else if (cMod == 2)
{
        int idxF = idxFtmp + c;
        int idxIn = idxIntmp + c;
        sum += pFilter[idxF]*pInput[idxIn];
        sum += pFilter[idxF+1]*pInput[idxIn+1];
}
```

Unrolled Kernel 2 (2)

```
            else if (cMod == 3)
            {
                  int idxF = idxFtmp + c;
                  int idxIn = idxIntmp + c;
                  sum += pFilter[idxF]*pInput[idxIn];
                  sum += pFilter[idxF+1]*pInput[idxIn+1];
                  sum += pFilter[idxF+2]*pInput[idxIn+2];
            }
      } //for (int r = 0…
      const int idxOut = yOut * nWidth + xOut;
      pOutput[idxOut] = sum;
}
```

Performance



Yet another way to achieve similar results is to write four different versions of the ConvolveUnroll kernel.
The four versions will correspond to (filterWidth%4) equalling 0, 1, 2, or 3.
The particular version called can be decided at run-time depending on the value of filterWidth

Kernel with Invariants

- Loop unrolling did not help when the filter width is low

- So far, kernels have been written in a generic way so that they will work for all filter sizes
- What if we can focus on a particular filter size?
  - E.g. 5×5. We can now unroll the inner loop five times and get rid of the loop condition
  - If we use the invariant in the loop condition, a good compiler will unroll the loop itself
  - `FILTER_WIDTH` can be passed to compiler

Kernel with Invariants

```
__kernel void Convolve_Def(const __global float * pInput,
      __constant float * pFilter, __global float * pOutput,
      const int nInWidth, const int nFilterWidth)
{
      const int nWidth = get_global_size(0);
      const int xOut = get_global_id(0);
      const int yOut = get_global_id(1);
      const int xInTopLeft = xOut;
      const int yInTopLeft = yOut;

      float sum = 0;
      for (int r = 0; r < FILTER_WIDTH; r++)
      {
            const int idxFtmp = r * FILTER_WIDTH;
            const int yIn = yInTopLeft + r;
            const int idxIntmp = yIn * nInWidth + xInTopLeft;
```

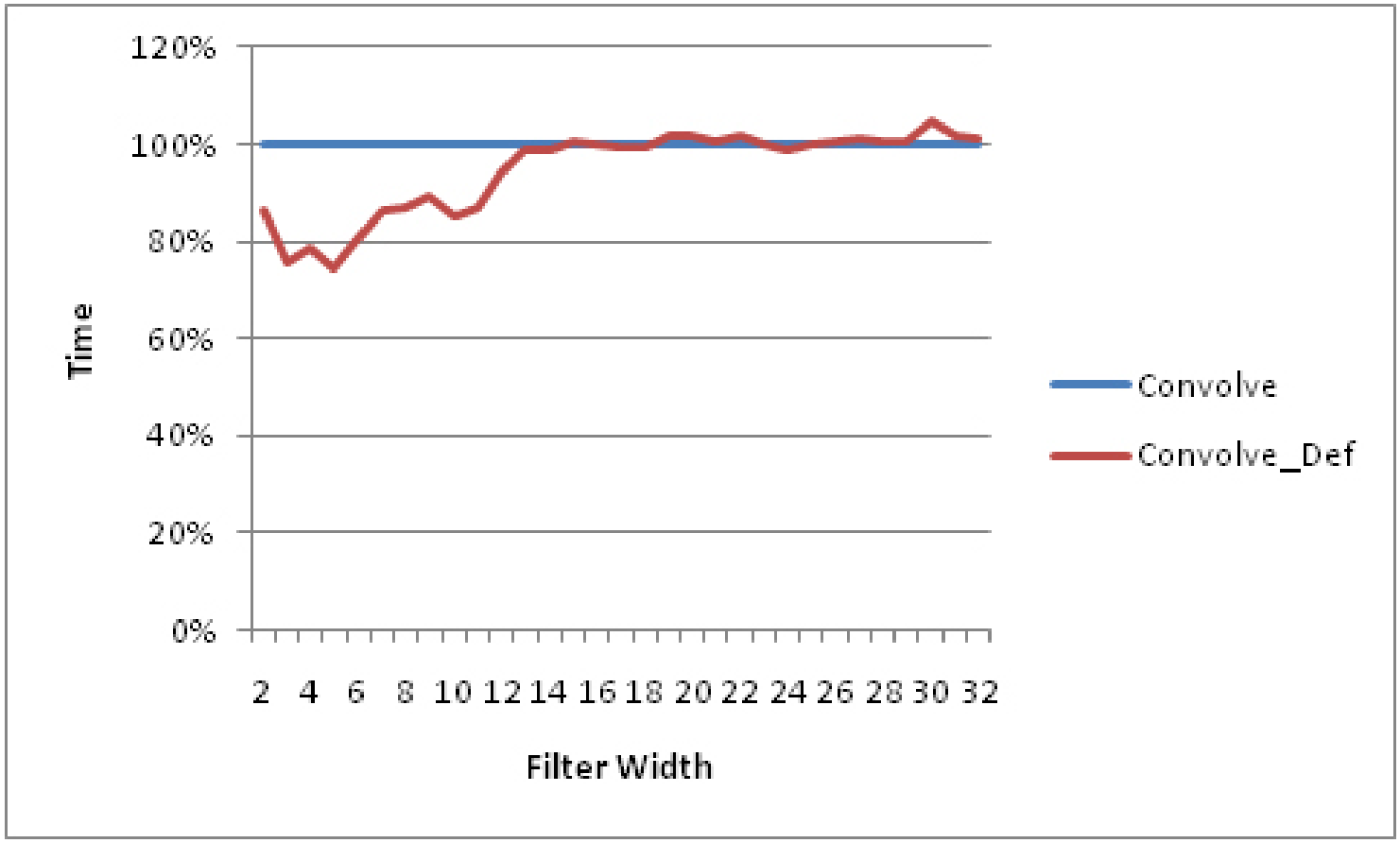
Kernel with Invariants (2)

```
            for (int c = 0; c < FILTER_WIDTH; c++)
            {
                  const int idxF = idxFtmp + c;
                  const int idxIn = idxIntmp + c;
                  sum += pFilter[idxF]*pInput[idxIn];
            }
      } //for (int r = 0…
      const int idxOut = yOut * nWidth + xOut;
      pOutput[idxOut] = sum;
}
```

Setting Filter Width

```
// this can be done online and offline

/* create a cl source string */
std::string sourceStr = Convert_File_To_String(File_Name);
cl::Program::Sources sources(1,
       std::make_pair(sourceStr.c_str(), sourceStr.length()));
/* create a cl program object */
program = cl::Program(context, sources);
/* build a cl program executable with some #defines */
char options[128];
sprintf(options, "-DFILTER_WIDTH=%d", filter_width);
program.build(devices, options);

/* create a kernel object for a kernel with the given name */
cl::Kernel kernel = cl::Kernel(program, "Convolve_Def");
```

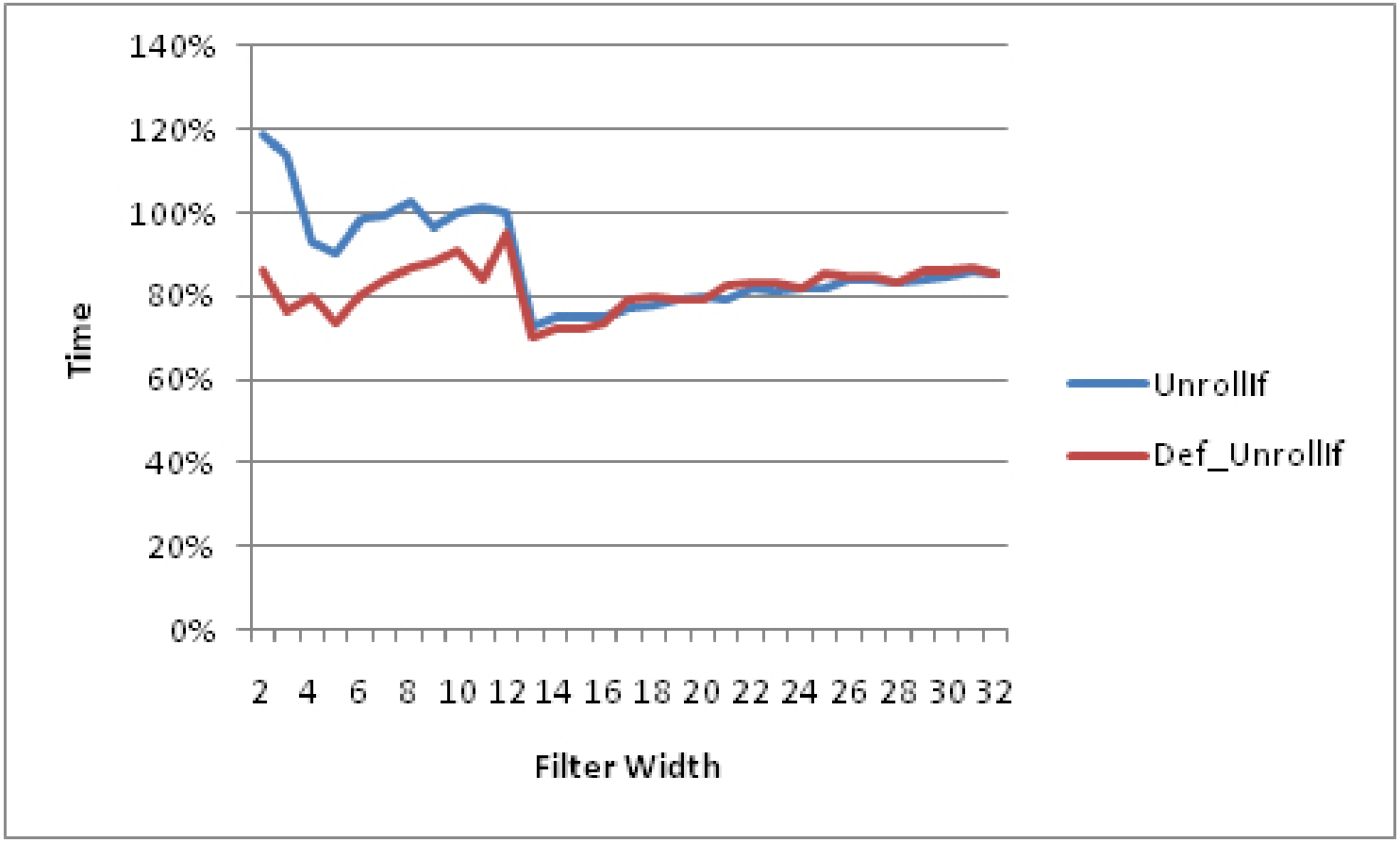
Performance



Performance

Performance

Unroll + if on remainder



Vectorization

```
__kernel void Convolve_Unroll(const __global float * pInput,
      __constant float * pFilter, __global float * pOutput,
      const int nInWidth, const int nFilterWidth)
{
      const int nWidth = get_global_size(0);
      const int xOut = get_global_id(0);
      const int yOut = get_global_id(1);
      const int xInTopLeft = xOut;
      const int yInTopLeft = yOut;

      float sum0 = 0; float sum1 = 0;
      float sum2 = 0; float sum3 = 0;

      for (int r = 0; r < nFilterWidth; r++)
      {
            const int idxFtmp = r * nFilterWidth;
```

Vectorization (2)

```
const int yIn = yInTopLeft + r;
const int idxIntmp = yIn * nInWidth + xInTopLeft;

int c = 0;
while (c <= nFilterWidth-4)
{
        float mul0, mul1, mul2, mul3;
        int idxF = idxFtmp + c;
        int idxIn = idxIntmp + c;
        mul0 = pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        mul1 += pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        mul2 += pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        mul3 += pFilter[idxF]*pInput[idxIn];
```

Vectorization (3)

```
                sum0 += mul0; sum1 += mul1;
                sum2 += mul2; sum3 += mul3;
                c += 4;
        }

        for (int c1 = c; c1 < nFilterWidth; c1++)
        {
                const int idxF = idxFtmp + c1;
                const int idxIn = idxIntmp + c1;
                sum0 += pFilter[idxF]*pInput[idxIn];
        }
    } //for (int r = 0…
    const int idxOut = yOut * nWidth + xOut;
    pOutput[idxOut] = sum0 + sum1 + sum2 + sum3;
}
```

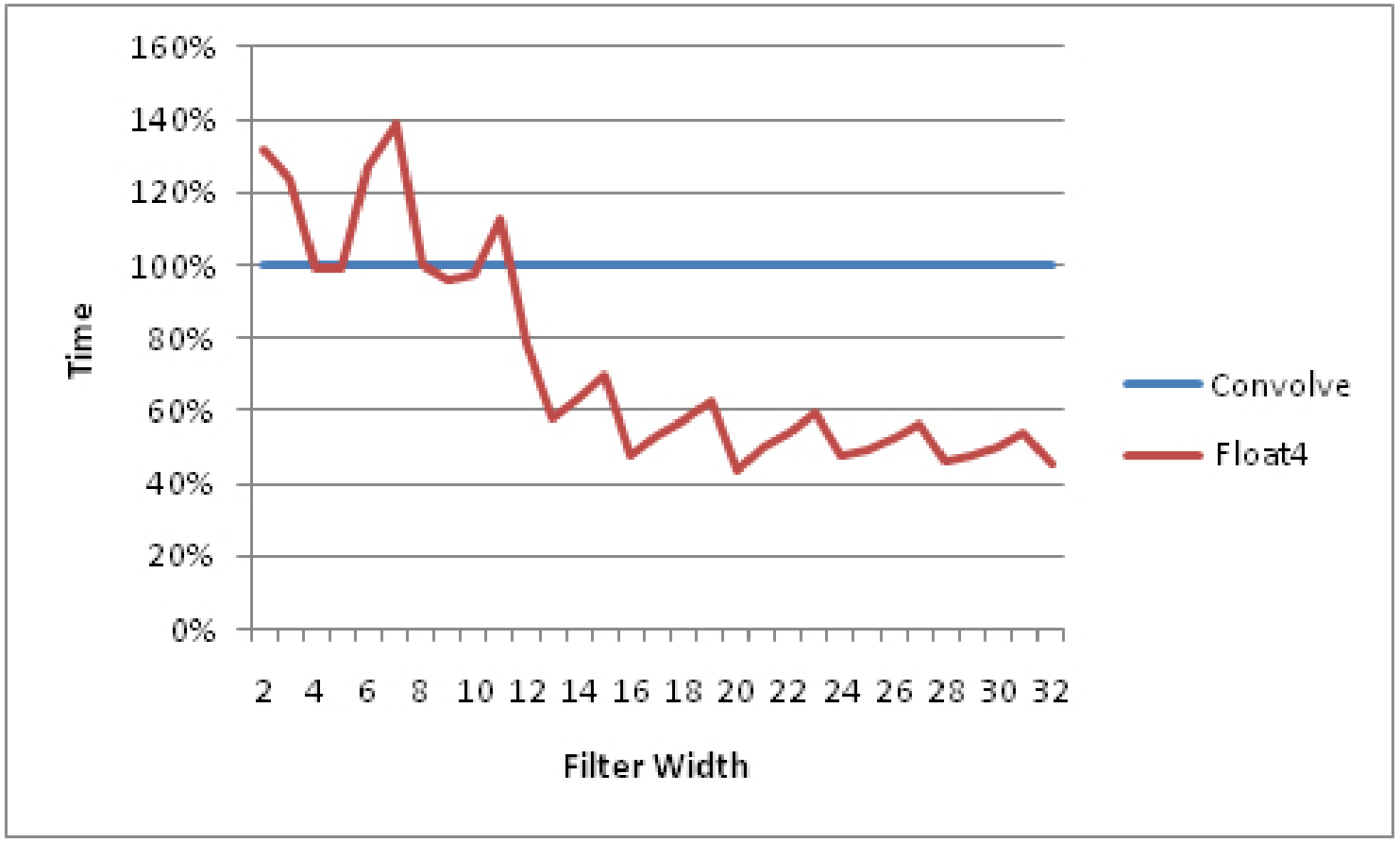
Vectorized Kernel

```
__kernel void Convolve_Float4(const __global float * pInput,
        __constant float * pFilter, __global float * pOutput,
        const int nInWidth, const int nFilterWidth)
{
        const int nWidth = get_global_size(0);
        const int xOut = get_global_id(0);
        const int yOut = get_global_id(1);
        const int xInTopLeft = xOut;
        const int yInTopLeft = yOut;

        float4 sum4 = 0;
        for (int r = 0; r < nFilterWidth; r++)
        {
                const int idxFtmp = r * nFilterWidth;
                const int yIn = yInTopLeft + r;
                const int idxIntmp = yIn * nInWidth + xInTopLeft;
```

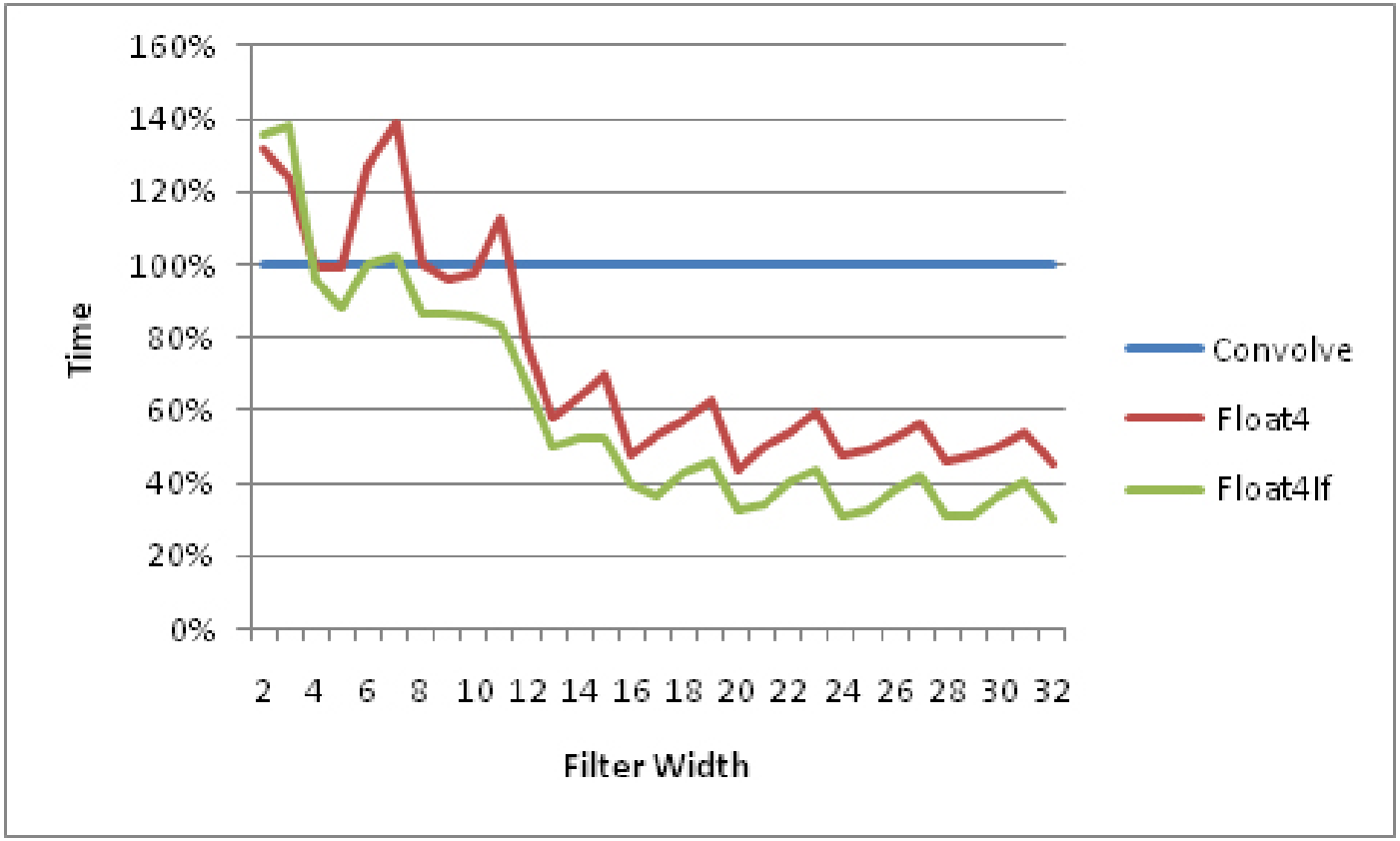
Vectorized Kernel

```
int c = 0; int c4 = 0;
while (c <= nFilterWidth-4)
{
        float4 filter4 = vload4(c4,pFilter+idxFtmp);
        float4 in4 = vload4(c4,pInput +idxIntmp);
        sum4 += in4 * filter4;
        c += 4;
        c4++;
}
for (int c1 = c; c1 < nFilterWidth; c1++) { const int idxF =
idxFtmp + c1; const int idxIn = idxIntmp + c1; sum4.x +=
pFilter[idxF]*pInput[idxIn]; } } //for (int r = 0…

const int idxOut = yOut * nWidth + xOut;
pOutput[idxOut] = sum4.x + sum4.y + sum4.z + sum4.w; }
```

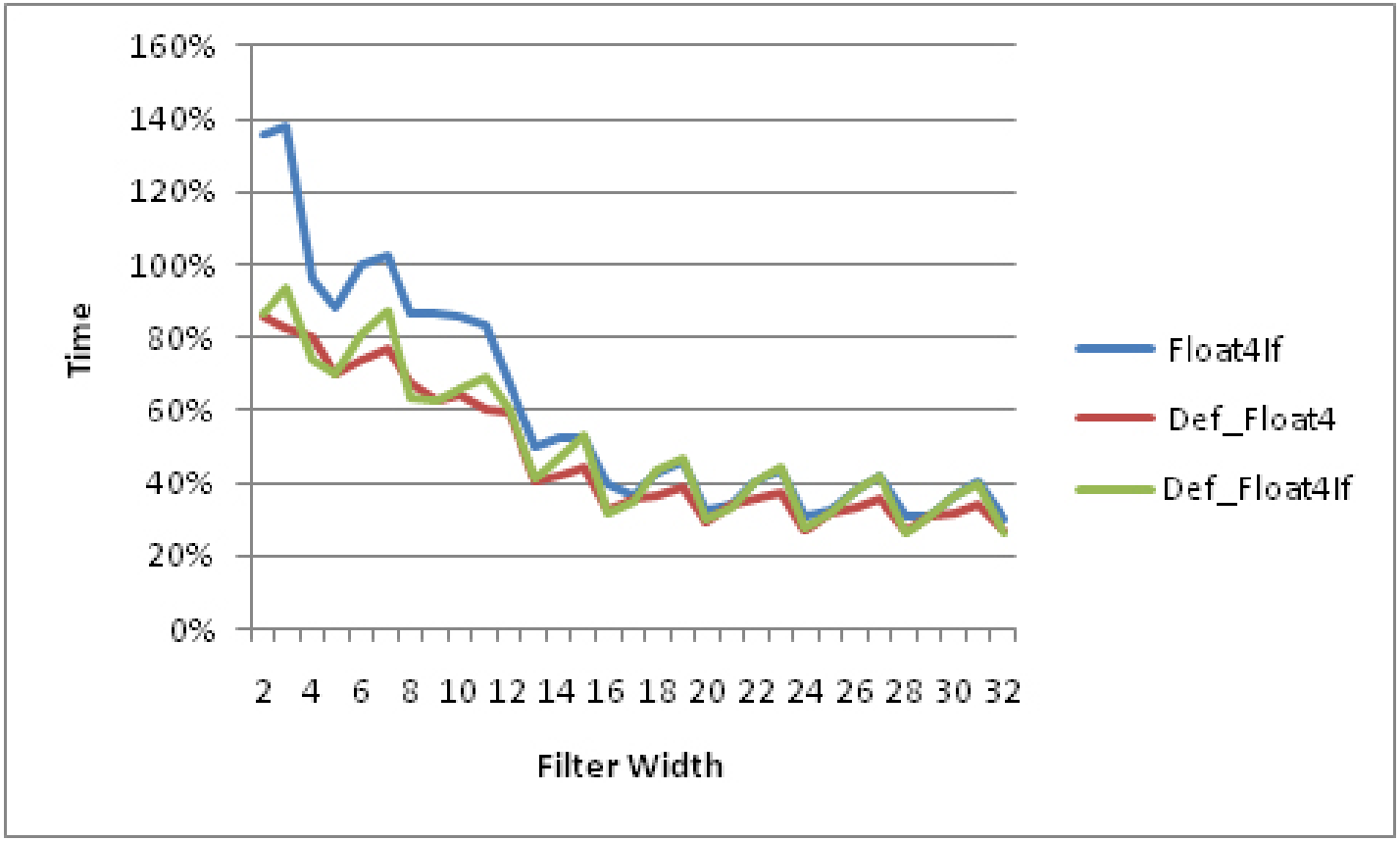
Performance



Performance – if Kernel

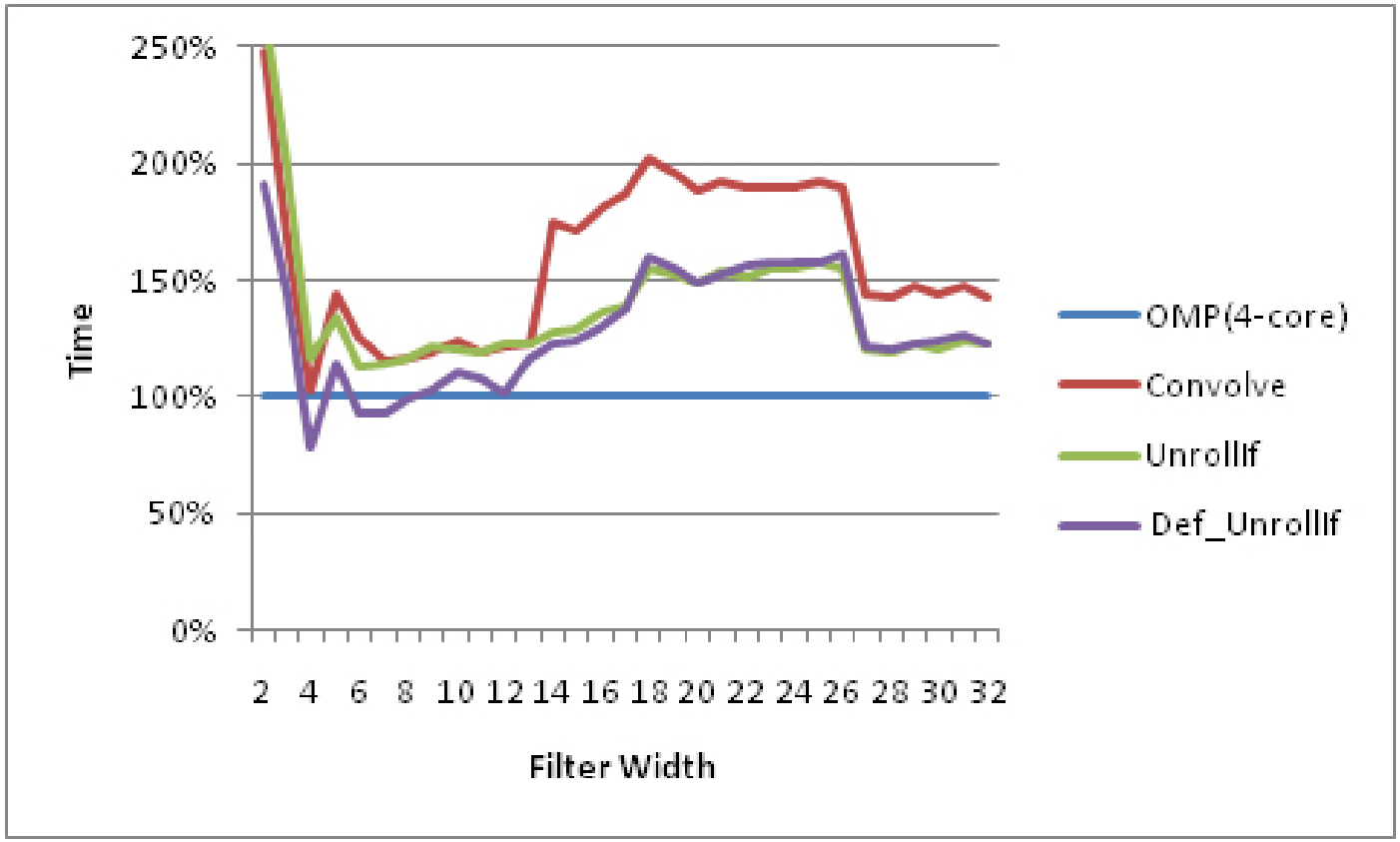


Performance – Kernel with Invariants

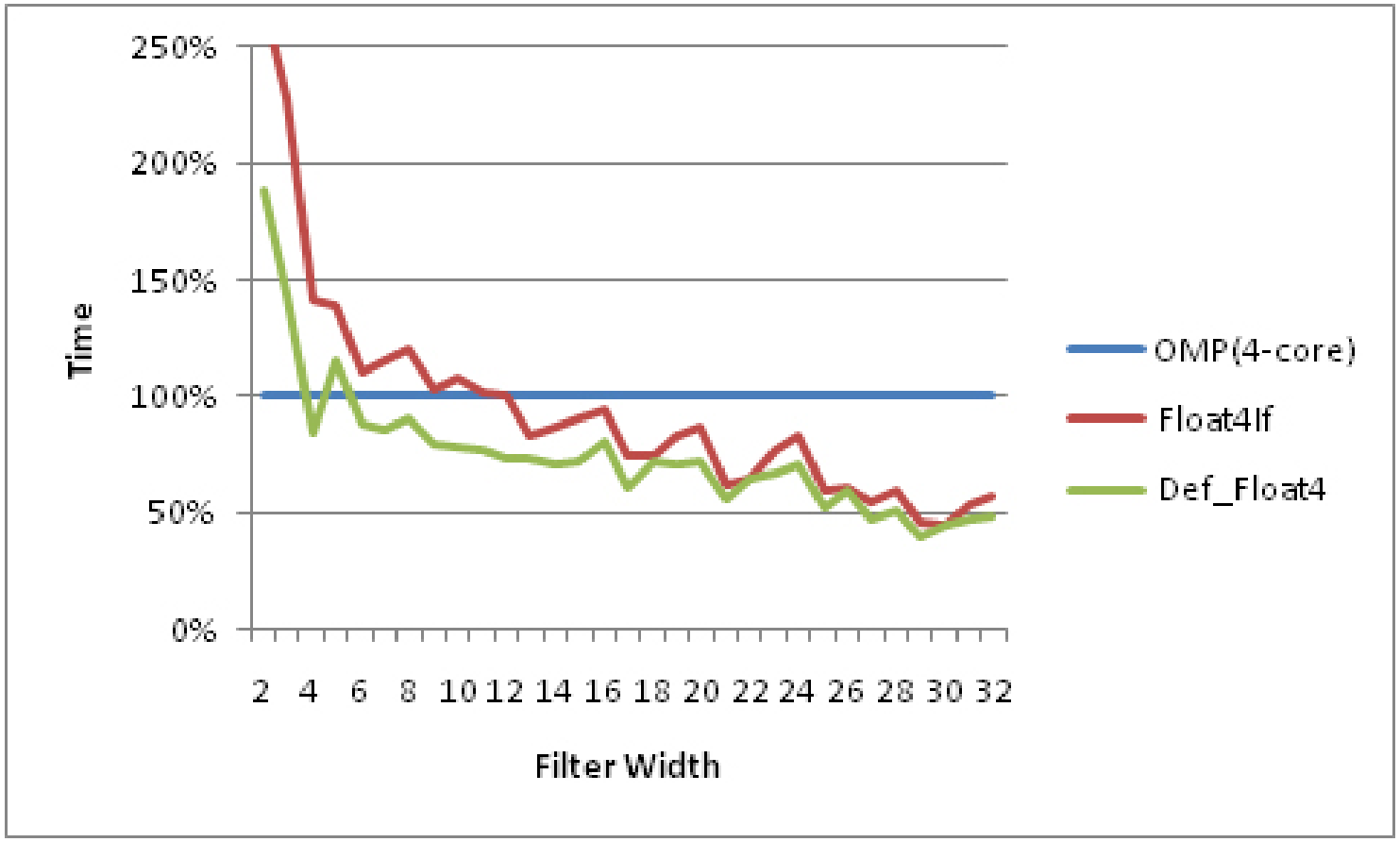


Instead of passing filterWidth as an argument to the kernel, we will define the value for FILTER_WIDTH when we build the OpenCL program object

OpenMP Comparison



OpenMP Comparison



Introduction to OpenACC Directives

Yonghong Yan

http://www.cs.uh.edu/~hpctools

University of Houston

Acknowledgements: Mark Harris (Nvidia), Duncan Pool (Nvidia)
Michael Wolfe (PGI), Sunita Chandrasekaran (UH), Barbara M. Chapman (UH)

OpenACC

- OpenACC API provides
  - compiler directives
  - library routines
  - environment variables
- Can be used to write data-parallel programs
  - FORTRAN
  - C/C++

OpenACC vs. CUDA

- OpenACC uses compiler directives

```
void saxpy(int n,
   float a,
   float *x,
   float *y)
{
   #pragma acc kernels
   for (int i = 0; i < n; ++i)
       y[i] = a*x[i] + y[i];
}
...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

OpenACC vs. CUDA

- Code almost identical to sequential version except for #pragma directives
  - #pragma provides to the compiler information not specified in standard language
- In OpenACC, you can write sequential program and then annotate it with directives
- Compiler takes care of data transfer, caching, kernel launching, parallelism mapping at runtime

OpenACC vs. CUDA

- OpenACC provides incremental path for moving legacy applications to accelerators
  - Disturbs existing code less than alternatives
- Non-OpenACC compiler ignores directives and generates sequential code
- Performance depends heavily on compiler that may not be able to follow some directives
- If directives are ignored, programs may give incorrect results

Jacobi Iteration

```
while ( err > tol && iter < iter_max ) {
   err=0.0;

   for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
              Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                   A[j-1][i] + A[j+1][i]);
              err = max(err, abs(Anew[j][i] - A[j][i]);
        }
   }

   for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
              A[j][i] = Anew[j][i];
        }
   }
iter++;
```

Iterate until convergence

Calculate new value from neighbors

Compute max error

Swap input/output arrays

Jacobi Iteration

```
while ( err > tol && iter < iter_max ) {
   err=0.0;

   #pragma acc kernels reduction(max:err)
   for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
                Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                     A[j-1][i] + A[j+1][i]);
                err = max(err, abs(Anew[j][i] - A[j][i]);
        }
   }

   #pragma acc kernels
   for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
        }
   }
iter++;
```

Execute GPU kernel for loop nest

Execute GPU kernel for loop nest

Loops vs Kernels

LOOP

```
for (int i=0; i<N; i++)
{
  C[i] = A[i] + B[i];
}
```

Calculate 0 - N in order

KERNEL

```
void loopBody(A,B,C,i)
{
  C[i] = A[i] + B[i];
}
```

Each compute core calculates one value of `i`.

Jacobi Iteration

```
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
   err=0.0;

   #pragma acc kernels reduction(max:err)
   for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
                Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                     A[j-1][i] + A[j+1][i]);
                err = max(err, abs(Anew[j][i] - A[j][i]);
        }
   }

   #pragma acc kernels
   for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
        }
   }
iter++;
```

Copy A in at the beginning of loop, out at the end. Allocate Anew on acc device

Structured Data Regions

- The data directive defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region. Here is an example of defining the structured data region with data directives:

```
#pragma acc data
{
   #pragma acc parallel loop ...
   #pragma acc parallel loop
   ...
}
```

Data Clauses

- **copyin(list)** - Allocates memory on GPU and copies data from host to GPU when entering region.
- **copyout(list)** - Allocates memory on GPU and copies data to the host when exiting region.
- **copy(list)** - Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region. (Structured Only)
- **create(list)** - Allocates memory on GPU but does not copy.
- **delete(list)** - Deallocate memory on the GPU without copying. (Unstructured Only)
- **present(list)** - Data is already present on GPU from another region containing data.

OpenACC Execution Model

- Host and accelerator device
  - Does not assume synchronization capability except fork and join

- Three levels: gang, worker, and vector
- Typical mapping for GPU:
  - gang==block
  - worker==warp
  - vector==threads of a warp

OpenACC Execution Model

- `Parallel` or `kernels` construct launches code on accelerator device
- `kernels` may contain sequence of kernels, each executed on accelerator device
- A group of gangs execute each kernel
- A group of workers is forked to execute a loop that belongs to a gang
  - Gang typically executes on one execution unit (SM)
  - Worker runs on one thread

OpenACC Execution Model

- If a `parallel` construct does not have an explicit `num_gangs` clause, then it is picked at runtime by implementation
  - Number of gangs and workers remain fixed during execution
- A `loop` construct is required for parallelizing a loop

Loop Examples

```
#pragma acc parallel num_gangs(1024)
{
   for (int i=0; i<2048;i++){
       …
   }
}
```

All gangs execute all iterations

```
#pragma acc parallel num_gangs(1024)
{
   # pragma acc loop gang
   for (int i=0; i<2048;i++){
       …
   }
}
```

Each gang lead assigned two iterations

Worker Loop Example

```
#pragma acc parallel num_gangs(1024)
  num_workers(32)
{
  # pragma acc loop gang
  for (int i=0; i<2048;i++){
     #pragma acc loop worker
     for (int j=0; j<512; j++){
           …
     }
  }
}
```

Each worker does 16 iterations (512/32) in each of the two outer iterations assigned to gang

Mapping OpenACC to CUDA Threads and Blocks

```
n = 128000;

#pragma acc kernels loop
for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

Uses whatever mapping to threads and blocks the compiler chooses

```
#pragma acc kernels loop gang(100), vector(128)
for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using kernels

```
#pragma acc parallel num_gangs(100), vector_length(128)
{
   #pragma acc loop gang, vector
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
}
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using parallel

Differences between kernels and parallel

- `kernels` is descriptive
  - Suggests to compiler which ultimately chooses based on performance and safety considerations
- `parallel` is prescriptive
  - Compiler does what user prescribes

Jacobi Iteration v. 2

```
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
   err=0.0;

   #pragma acc kernels reduction(max:err)
   for( int j = 1; j < n-1; j++) {
        #pragma acc loop gang(16) vector(32)
        for(int i = 1; i < m-1; i++) {
                Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                     A[j-1][i] + A[j+1][i]);
                err = max(err, abs(Anew[j][i] - A[j][i]);
        }
   }

   #pragma acc kernels
   for( int j = 1; j < n-1; j++) {
        #pragma acc loop gang(16) vector(32)
        for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
        }
   }
iter++;
```

Specify width of grids (16) and of blocks (32), but let compiler pick height

The present Clause

```
function main(int argc, char **argv) {
#pragma acc data copy(A) {
    laplace2D(A,n,m);
}
}
...
function laplace2D(double[N][M] A,n,m){
   #pragma acc data present(A[n][m]) create(Anew)
   while ( err > tol && iter < iter_max ) {
      err=0.0;
      ...
   }
}
```

OpenMP

Based on tutorial by Joel Yliluoma

http://bisqwit.iki.fi/story/howto/openmp/

OpenMP in C++

- OpenMP consists of a set of compiler #pragmas that control how the program works.
- The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.

Simple Example

- Multiple threads

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

Simple Example

- Single thread multiple data, SIMD

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp simd
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

Simple Example

- Multiple threads on another device

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

   #pragma omp target teams distribute parallel for
           map(from:sinTable[0:256])
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

Syntax

- All OpenMP constructs start with `#pragma omp`
- The `parallel` construct
  - Creates a *team* of N threads (N determined at runtime) all of which execute statement or next block
  - All variables declared within block become local variables to each thread
  - Variables shared from the context are handled transparently, sometimes by passing a reference and sometimes by using register variables

if

```
extern int parallelism_enabled;
 #pragma omp parallel for if(parallelism_enabled)
 for(int c=0; c<n; ++c)
   handle(c);
```

for

```
#pragma omp for
 for(int n=0; n<10; ++n)
 {
   printf(" %d", n);
 }
 printf(".\n");
```

- Output may appear in arbitrary order

Creating a New Team

```
#pragma omp parallel
 {
  #pragma omp for
  for(int n=0; n<10; ++n) printf(" %d", n);
 }
 printf(".\n");
```

- Or, equivalently

```
#pragma omp parallel for
 for(int n=0; n<10; ++n) printf(" %d", n);
 printf(".\n");
```

Specifying Number of Threads

```
#pragma omp parallel num_threads(3)
 {
   // This code will be executed by three threads.

   // Chunks of this loop will be divided amongst
   // the (three) threads of the current team.
   #pragma omp for
   for(int n=0; n<10; ++n) printf(" %d", n);
 }
```

`parallel, for, parallel for`

The difference between `parallel`, `parallel for` and `for` is as follows:

- A team is the group of threads that execute concurrently.
  - At the program beginning, the team consists of a single thread.
  - A `parallel` construct splits the current thread into a new team of threads for the duration of the next block/statement, after which the team merges back into one.
- `for` divides the work of the for-loop among the threads of the current team. It does not create threads.
- `parallel for` is a shorthand for the two commands at once. `Parallel` creates a new team, and `for` splits that team to handle different portions of the loop.
- If your program never contains a parallel construct, there is never more than one thread.

Scheduling

- Each thread independently decides which chunk of the loop it will process

```
#pragma omp for schedule(static)
 for(int n=0; n<10; ++n) printf(" %d", n);
 printf(".\n");
```

- In dynamic scheduling, each thread asks the OpenMP runtime library for an iteration number, handles it and asks for the next.
  - Useful when different iterations take different amounts of time to execute

```
#pragma omp for schedule(dynamic)
 for(int n=0; n<10; ++n) printf(" %d", n);
 printf(".\n");
```

Scheduling

- Each thread asks for iteration number, executes 3 iterations, then asks for another

```
#pragma omp for schedule(dynamic, 3)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

ordered

```
#pragma omp for ordered schedule(dynamic)
for(int n=0; n<100; ++n)
{
  files[n].compress();

  #pragma omp ordered
  send(files[n]);
}
```

reduction

```
 int sum=0;
#pragma omp parallel for reduction(+:sum)
for(int n=0; n<1000; ++n)
          sum += table[n];
```

Sections

```
#pragma omp parallel sections
{
  { Work1(); }
  #pragma omp section
  { Work2();
    Work3(); }
  #pragma omp section
  { Work4(); }
}
```

- Sections executed in parallel
- A barrier is implicitly defined at the end of the larger program region associated with the **omp sections** directive unless the **nowait** clause is specified

```
#pragma omp parallel // starts a new team
{
  Work0(); // this function would be run by all threads.

  #pragma omp sections // divides the team into sections
  {
    // everything herein is run only once.
    { Work1(); }
    #pragma omp section
    { Work2();
      Work3(); }
    #pragma omp section
    { Work4(); }
  }

  Work5(); // this function would be run by all threads.
}
```

simd

- SIMD means that multiple calculations will be performed simultaneously using special instructions that perform the same calculation to multiple values at once.
- This is often more efficient than regular instructions that operate on single data values. This is also sometimes called vector parallelism or vector operations.

```
float a[8], b[8];
 ...
 #pragma omp simd
 for(int n=0; n<8; ++n) a[n] += b[n];
```

simd

```
#pragma omp declare simd aligned(a,b:16)
  void add_arrays(float *__restrict__ a, float
*__restrict__ b)
  {
    #pragma omp simd aligned(a,b:16)
    for(int n=0; n<8; ++n) a[n] += b[n];
  }
```

Reduction:

```
int sum=0;
 #pragma omp simd reduction(+:sum)
 for(int n=0; n<1000; ++n) sum += table[n];
```

aligned

```
 #pragma omp declare simd aligned(a,b:16)
  void add_arrays(float *__restrict__ a, float
*__restrict__ b)
  {
    #pragma omp simd aligned(a,b:16)
    for(int n=0; n<8; ++n) a[n] += b[n];
  }
```

- Tells compiler that each element is aligned to the given number of bytes
- Increases performance

declare target

```
 #pragma omp declare target
int x;
void murmur() { x+=5; }
#pragma omp end declare target
```

- This creates one or more versions of "x" and "murmur". A set that exists on the host computer, and also a separate set that exists and can be run on a device.
- These two functions and variables are separate, and may contain values separate from each other's.

target, target data

- The target data construct creates a device data environment.
- The target construct executes the construct on a device (and also has target data features).
- These two constructs are identical in effect:

```
 #pragma omp target // device()... map()... if()...
{
  <<statements...>>
}
.....
#pragma omp target data // device()... map()... if()...
{
  #pragma omp target
  {
    <<statements...>>
  }
}
```

critical

- Restricts the execution of the associated statement / block to a single thread at a time
- May optionally contain a global name that identifies the type of the critical construct. No two threads can execute a critical construct of the same name at the same time.

- Below, only one of the critical sections named "dataupdate" may be executed at any given time, and only one thread may be executing it at that time. I.e. the functions "reorganize" and "reorganize_again" cannot be invoked at the same time, and two calls to the function cannot be active at the same time

```
#pragma omp critical(dataupdate)
{
  datastructure.reorganize();
}
...
#pragma omp critical(dataupdate)
{
  datastructure.reorganize_again();
}
```

private, firstprivate, shared

```
int a, b=0;
#pragma omp parallel for private(a) shared(b)
for(a=0; a<50; ++a)
{
  #pragma omp atomic
  b += a;
}
```

private, firstprivate, shared

- Variables with static storage duration are shared.
- Dynamically allocated objects are shared.
- Variables with automatic storage duration that are declared in a parallel region are private.
- Variables in heap allocated memory are shared. There can be only one shared heap.
- All variables defined outside a parallel construct become shared when the parallel region is encountered.
- Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
- Memory allocated within a parallel loop by the alloca function persists only for the duration of one iteration of that loop and is private for each thread.

private, firstprivate, shared

```
#include <string>
#include <iostream>

int main()
{
    std::string a = "x", b = "y";
    int c = 3;

    #pragma omp parallel private(a,c) shared(b)
            num_threads(2)
    {
        a += "k";
        c += 7;
        std::cout << "A becomes (" << a << "),
                  b is (" << b << ")\n";
    }
}
```

- Outputs “k” not “xk”, c is uninitialized

private, firstprivate, shared

```
#include <string>
#include <iostream>

int main()
{
    std::string a = "x", b = "y";
    int c = 3;

    #pragma omp parallel firstprivate(a,c) shared(b)
            num_threads(2)
    {
        a += "k";
        c += 7;
        std::cout << "A becomes (" << a << "),
                  b is (" << b << ")\n";
    }
}
```

- Outputs “xk”

Barriers

```
#pragma omp parallel
 {
   /* All threads execute this. */
   SomeCode();

   #pragma omp barrier

   /* All threads execute this, but not before
    * all threads have finished executing
      SomeCode().
    */
   SomeMoreCode();
 }
```

```
#pragma omp parallel
 {
   #pragma omp for
   for(int n=0; n<10; ++n) Work();

   // This line is not reached before the for-loop is completely finished
   SomeMoreCode();
 }

 // This line is reached only after all threads from
 // the previous parallel block are finished.
 CodeContinues();

 #pragma omp parallel
 {
   #pragma omp for nowait
   for(int n=0; n<10; ++n) Work();

   // This line may be reached while some threads are still executing for-loop.
   SomeMoreCode();
 }

 // This line is reached only after all threads from
 // the previous parallel block are finished.
 CodeContinues();
```

Nested Loops

```
#pragma omp parallel for
 for(int y=0; y<25; ++y)
 {
   #pragma omp parallel for
   for(int x=0; x<80; ++x)
   {
     tick(x,y);
   }
 }
```

- Code above fails, inner loop runs is sequence

```
#pragma omp parallel for collapse(2)
for(int y=0; y<25; ++y)
  for(int x=0; x<80; ++x)
  {
    tick(x,y);
  }
```