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

Instructor: Philippos Mordohai
Webpage: www.cs.stevens.edu/~mordohai
E-mail: Philippos.Mordohai@stevens.edu

Final Project Presentations

- May 3
  - **Submit PPT/PDF file by 4pm**
  - 9 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 against CPU version

Final Report

- Due May 10 (11:59pm)
- 6-10 pages including figures, tables and references
- Content
  - See presentation instructions
  - Do not repeat course material
- Counts for 15% of total grade
- NO LATE SUBMISSIONS

Outline

- OpenCL Convolution Example
- Parallel sorting
- OpenACC

Image Convolution Using OpenCL™

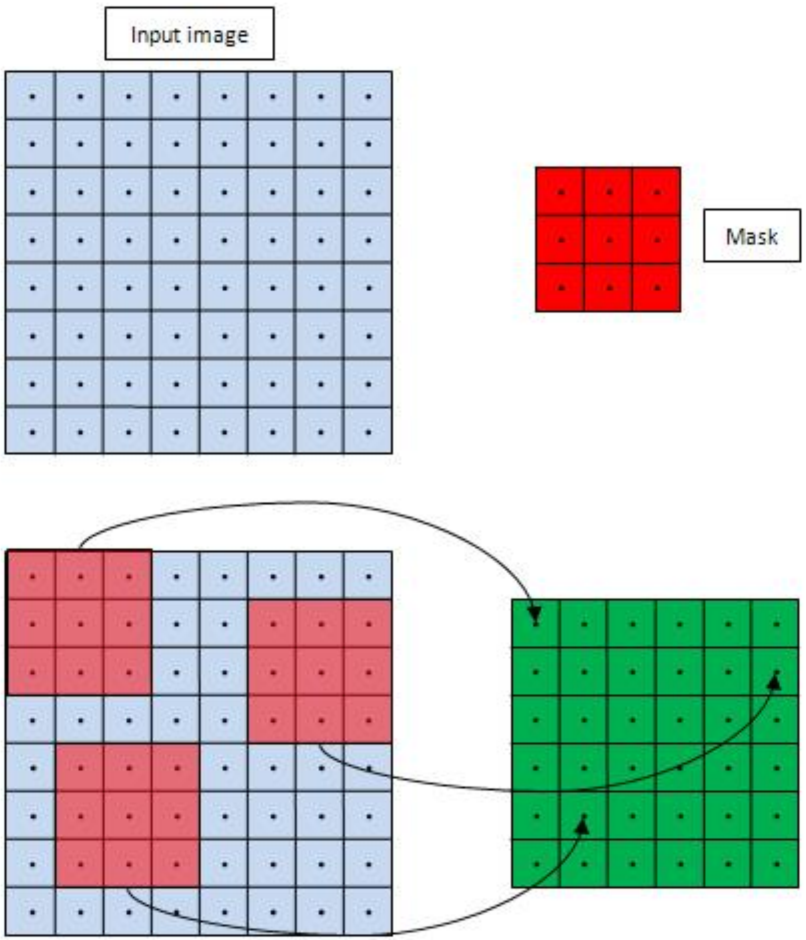
Udeepta Bordoloi,
ATI Stream Application Engineer

10/13/2009

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

http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/programming-in-opencl/image-convolution-using-opencl/

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);
```

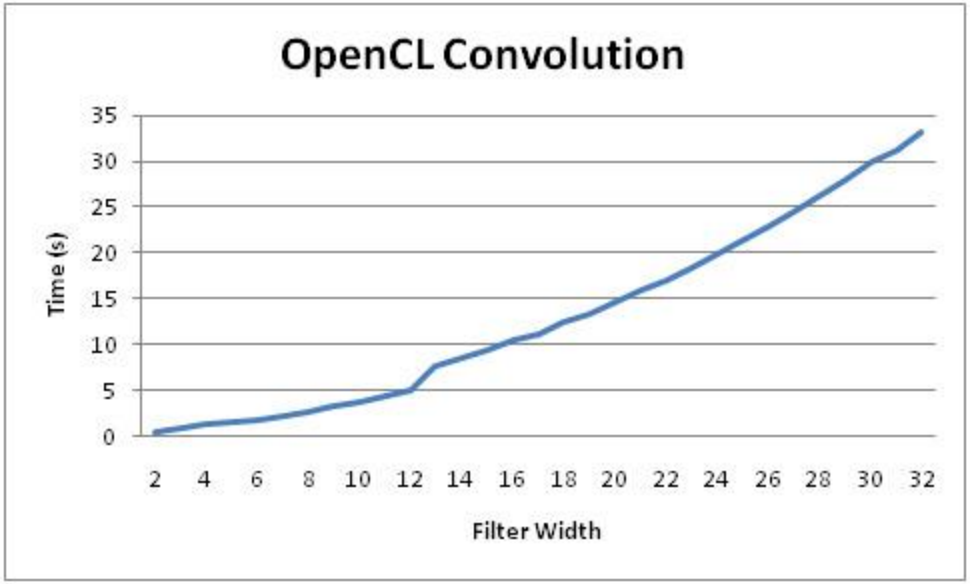
Execute Kernel

```
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;
```

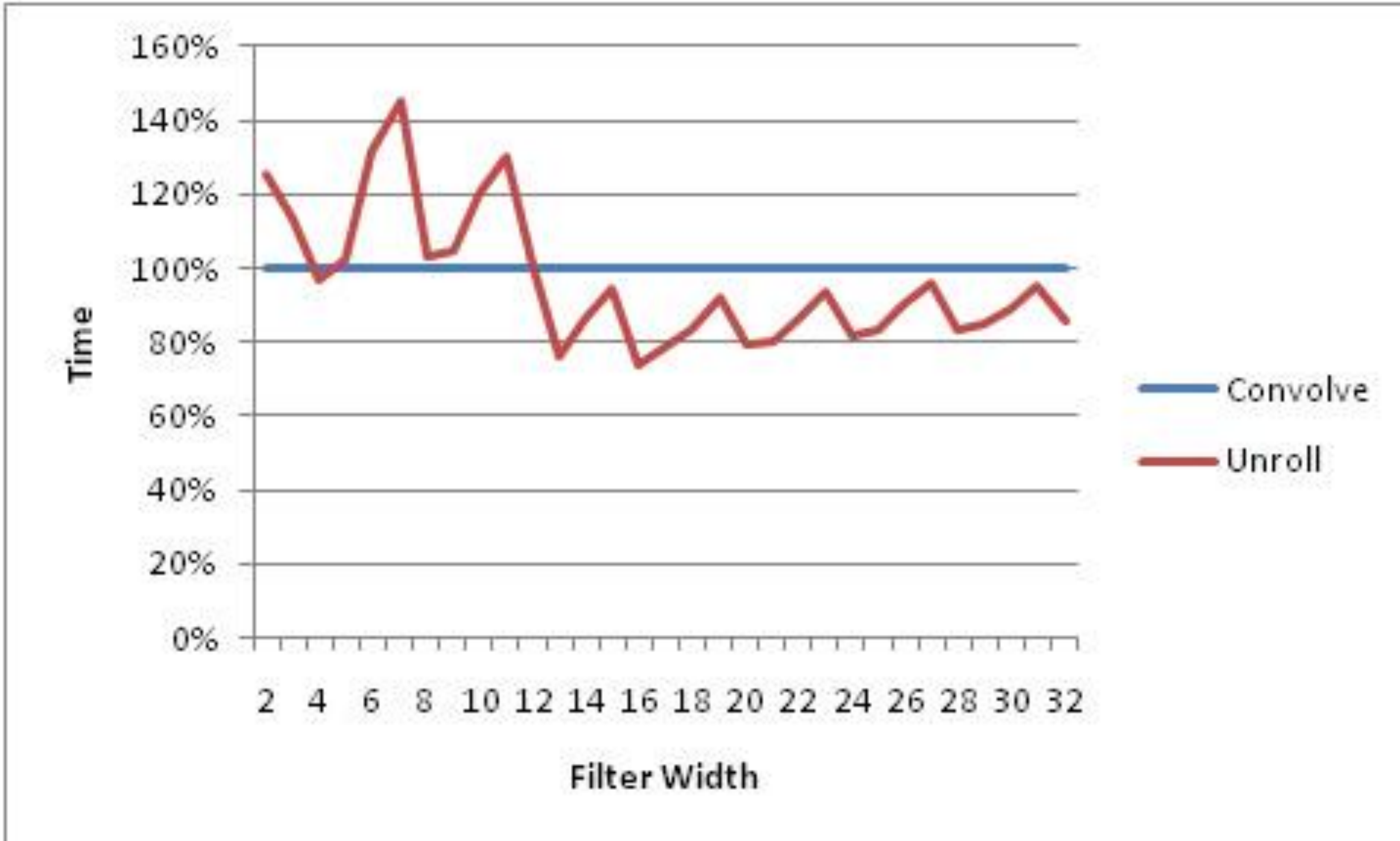
Loop Unrolling (2)

```
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



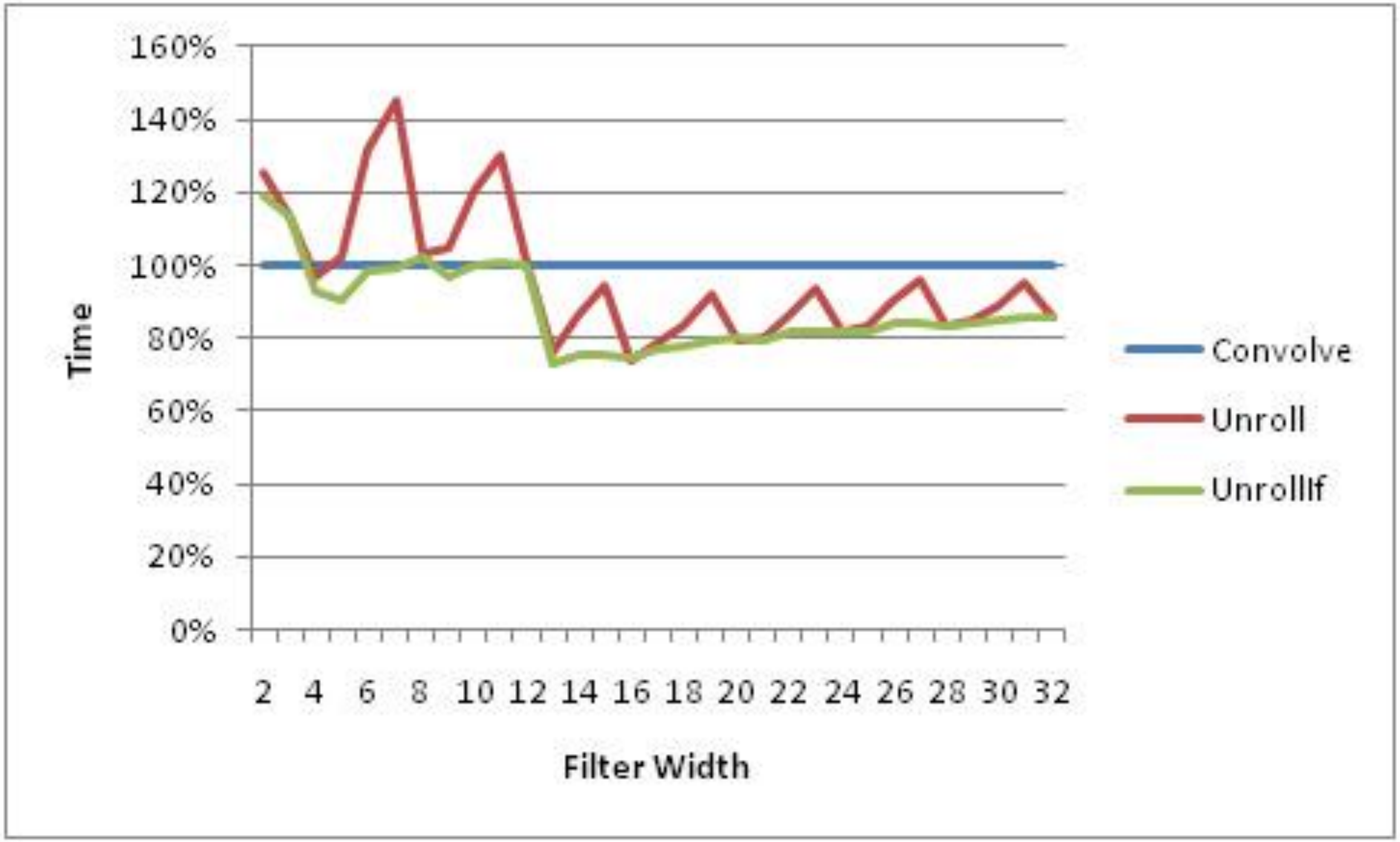
Unrolled Kernel 2 (if Kernel)

```
// 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;
```

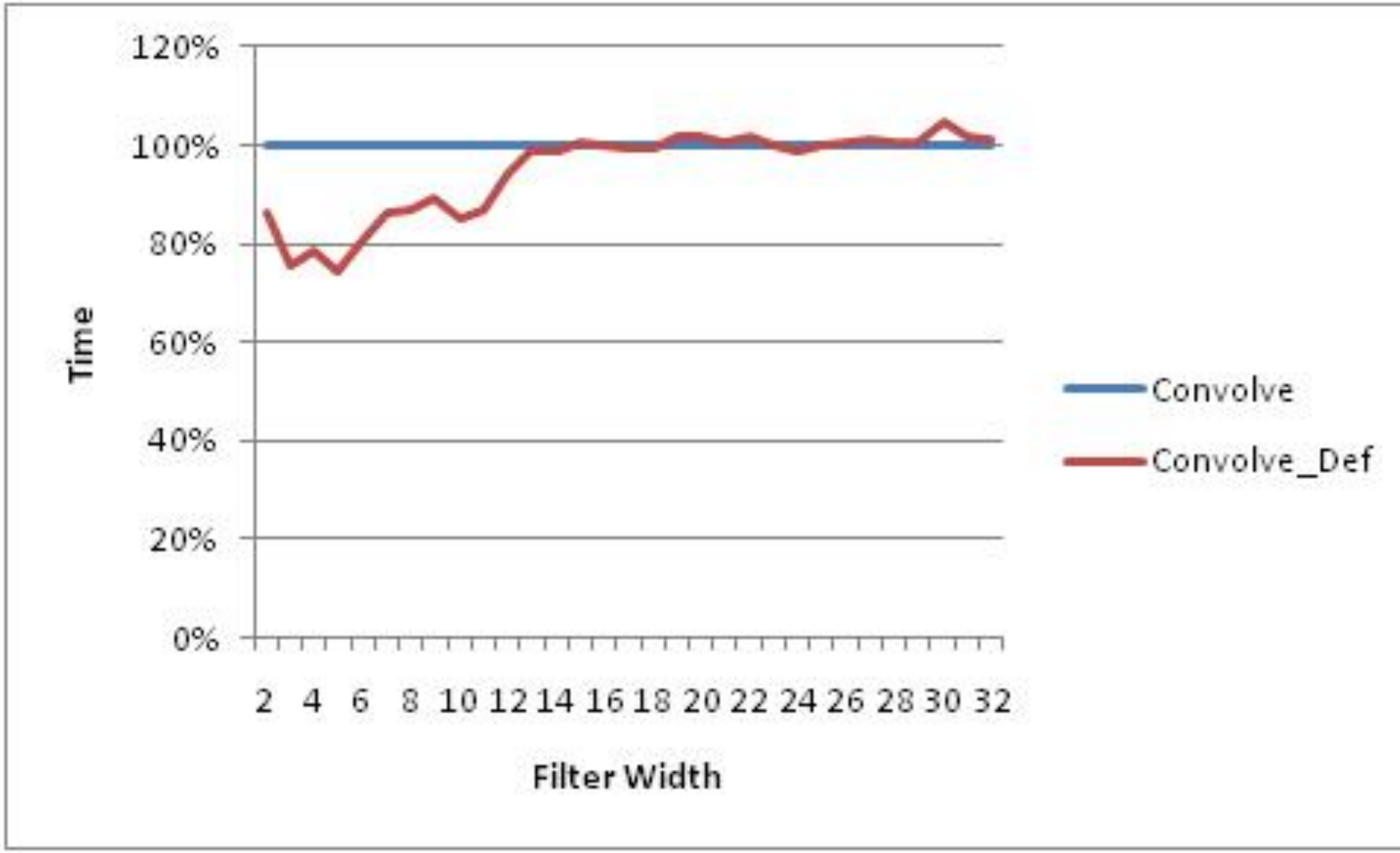
Kernel with Invariants (2)

```
        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;
}
```

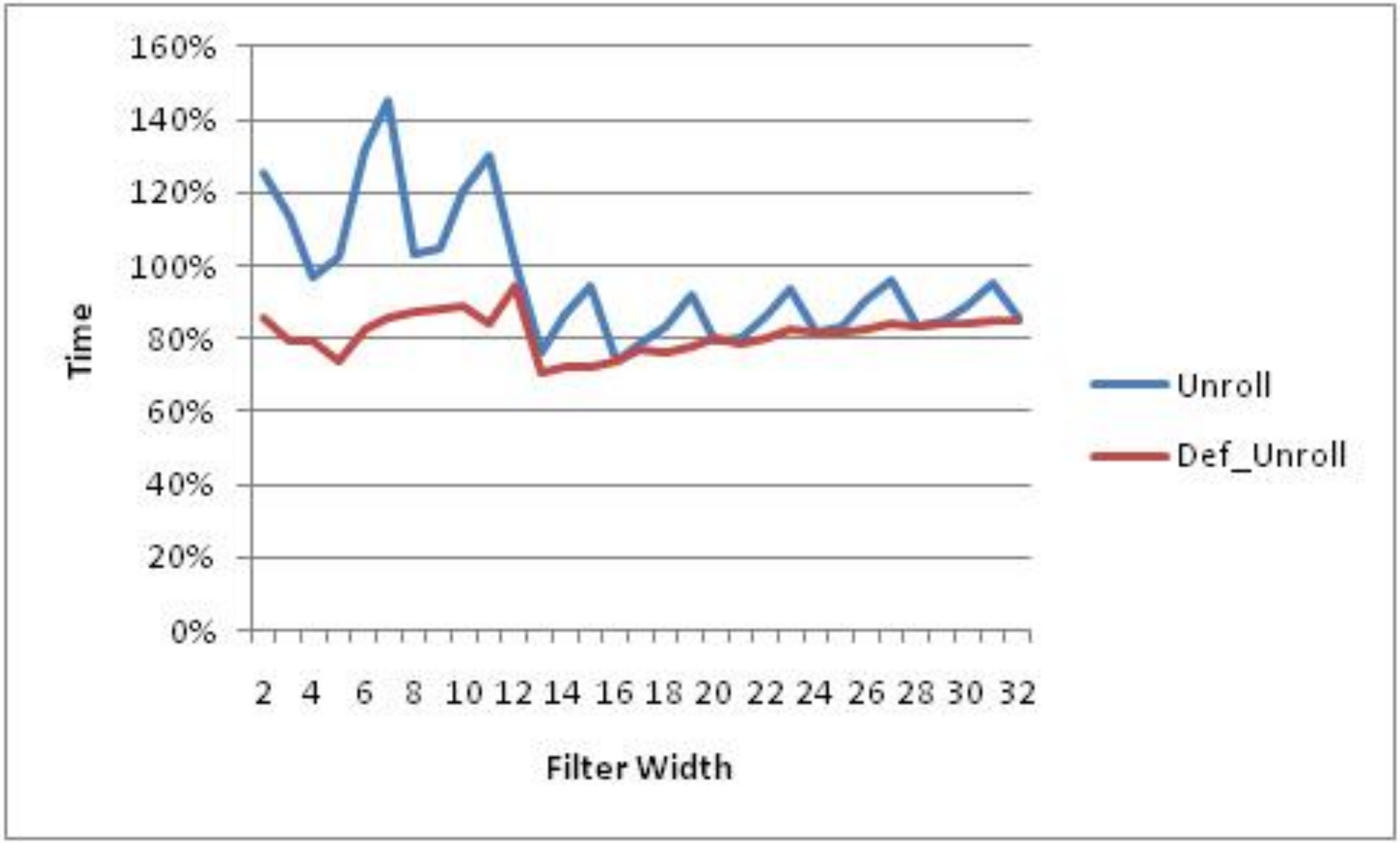
Setting Filter Width

```
// 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

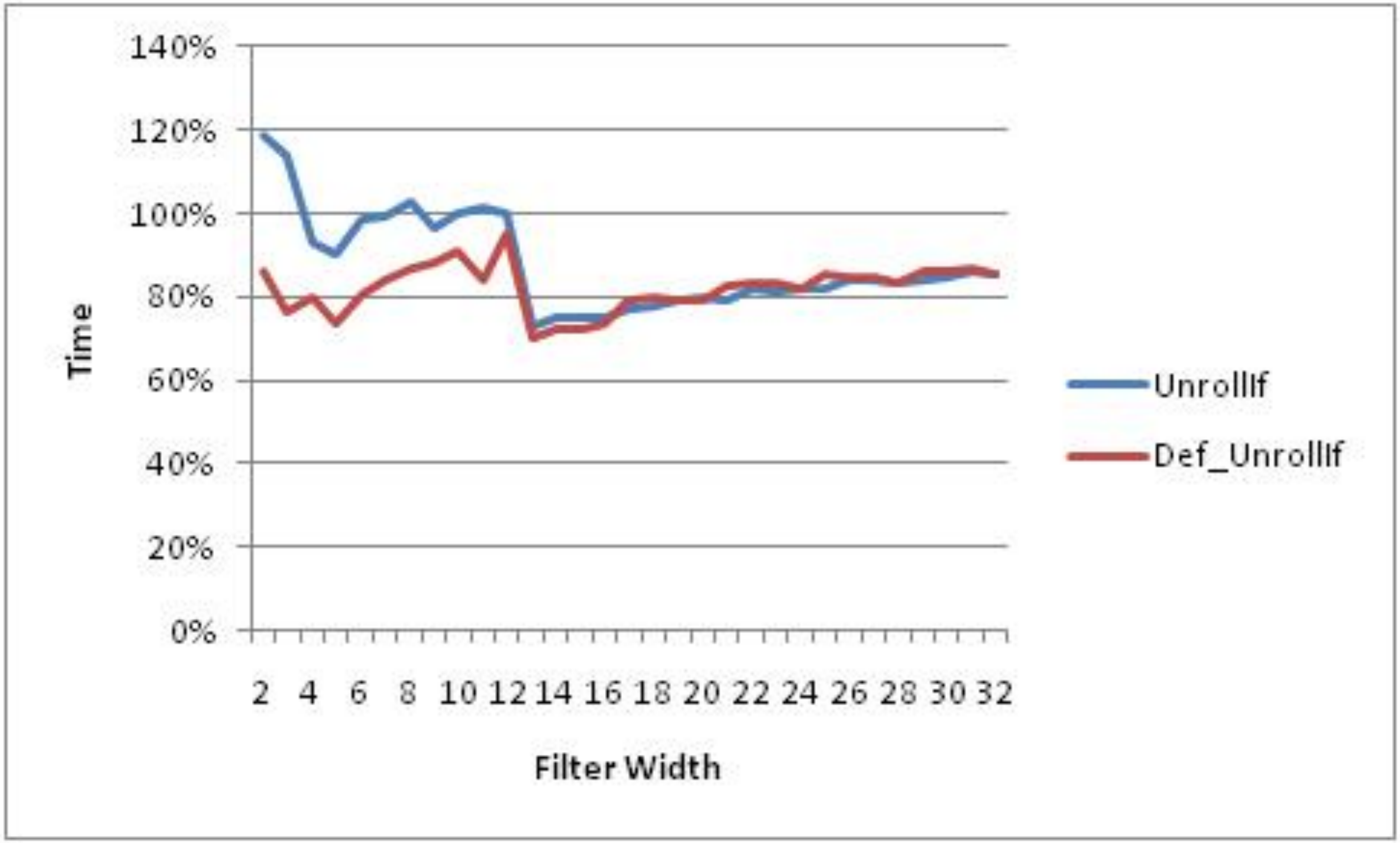


Performance



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;
}
```

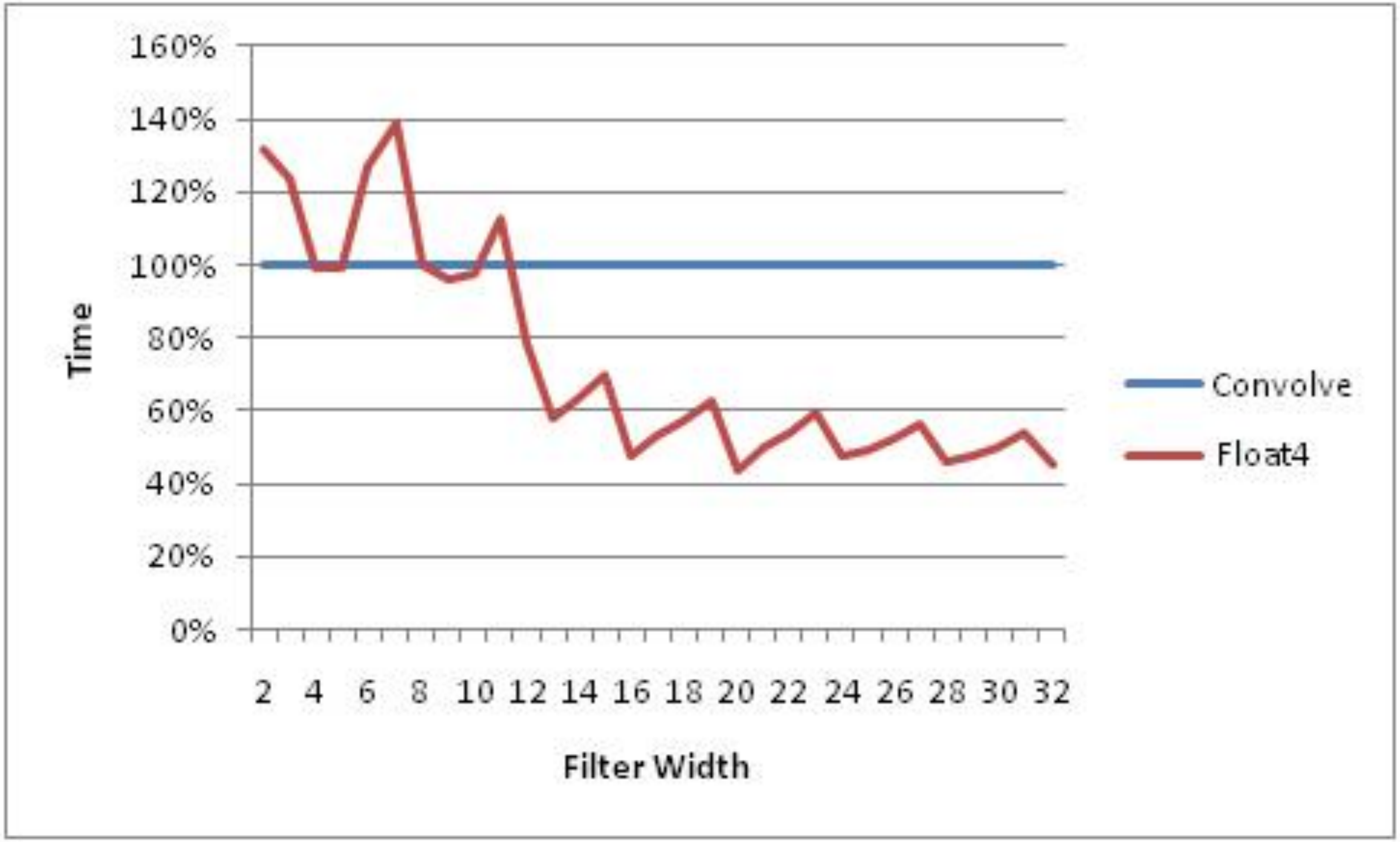
Vectorized Kernel

```
__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;
```

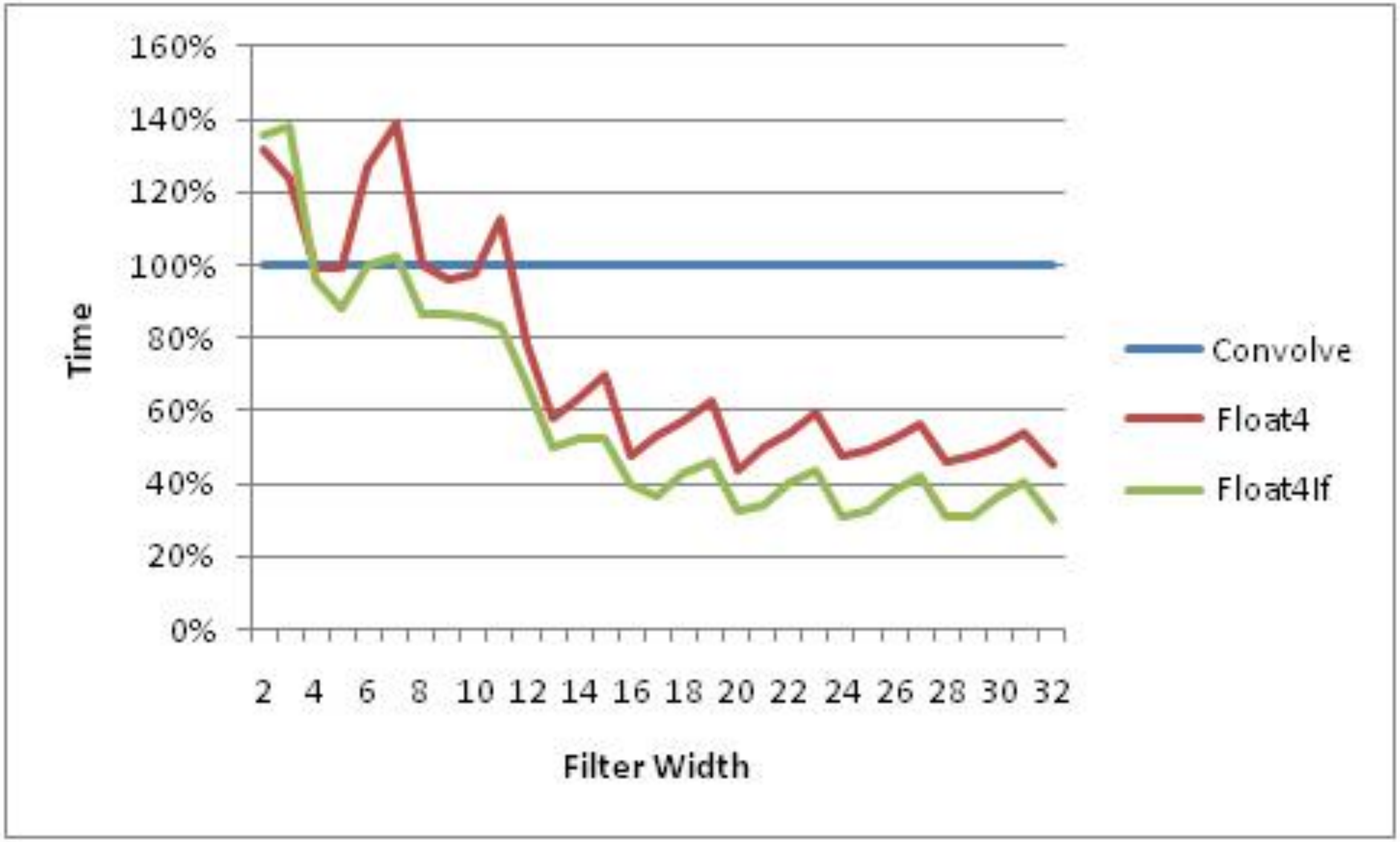
Vectorized Kernel

```
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; }
```

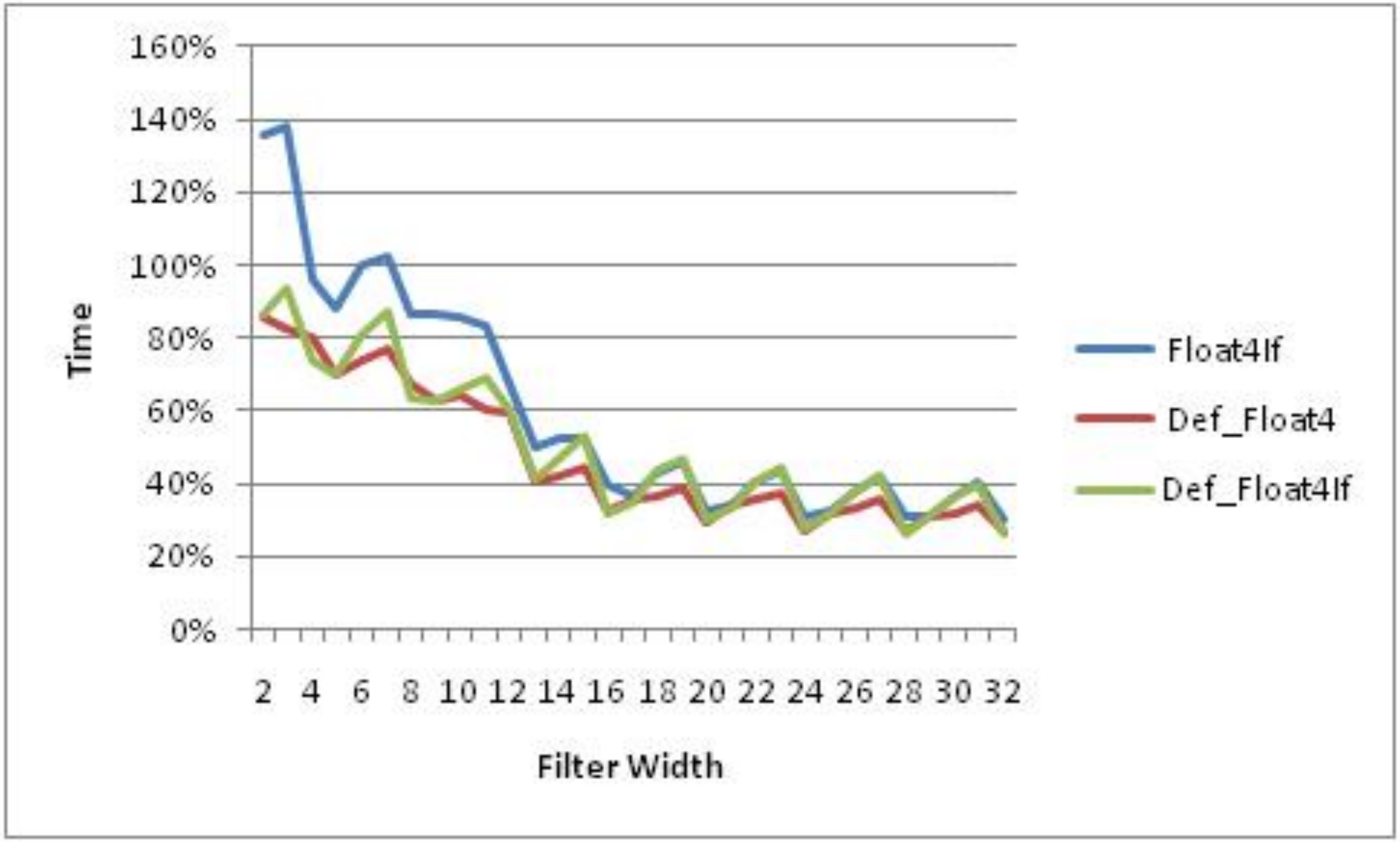
Performance



Performance – if Kernel

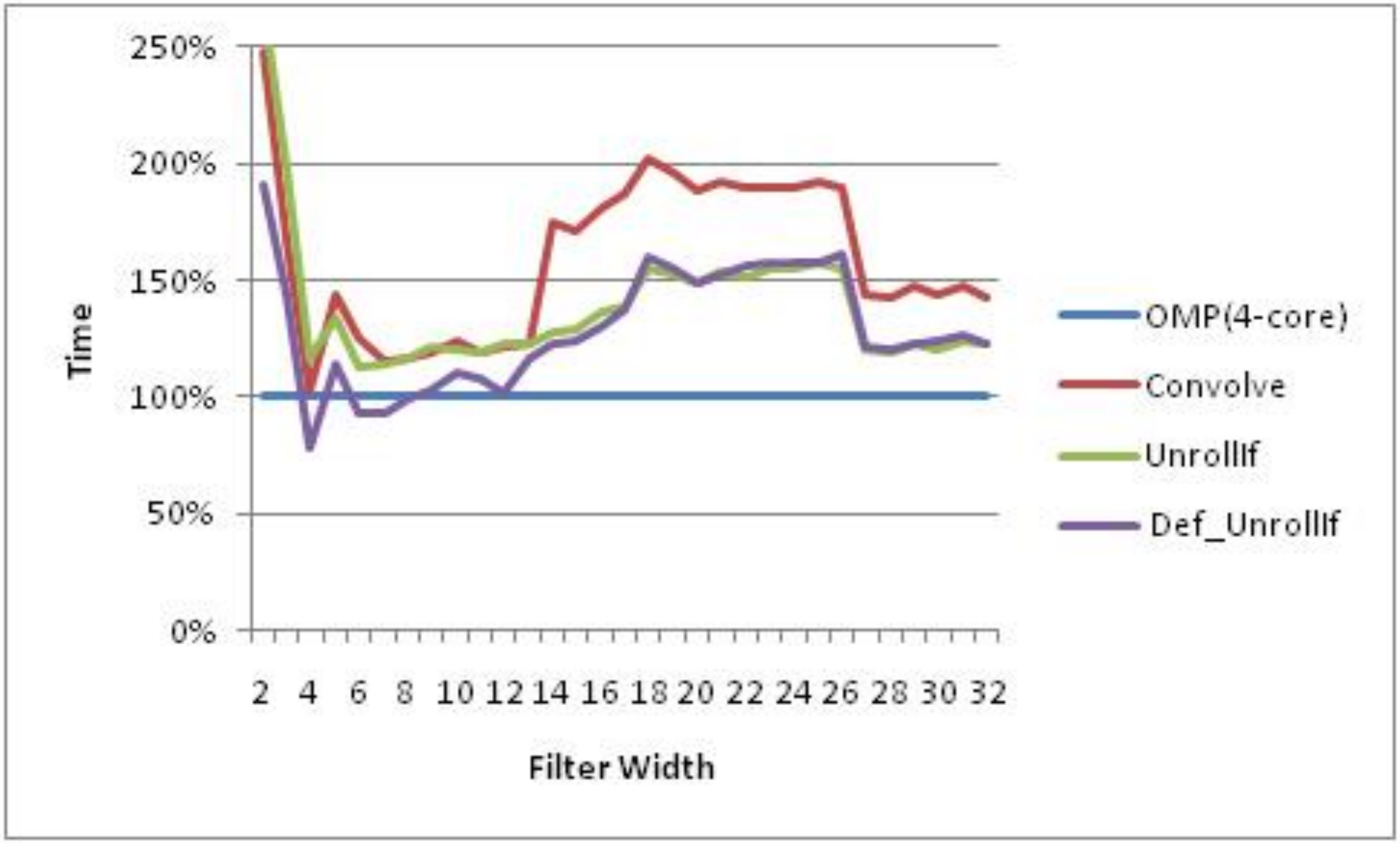


Performance - Kernel with Invariants

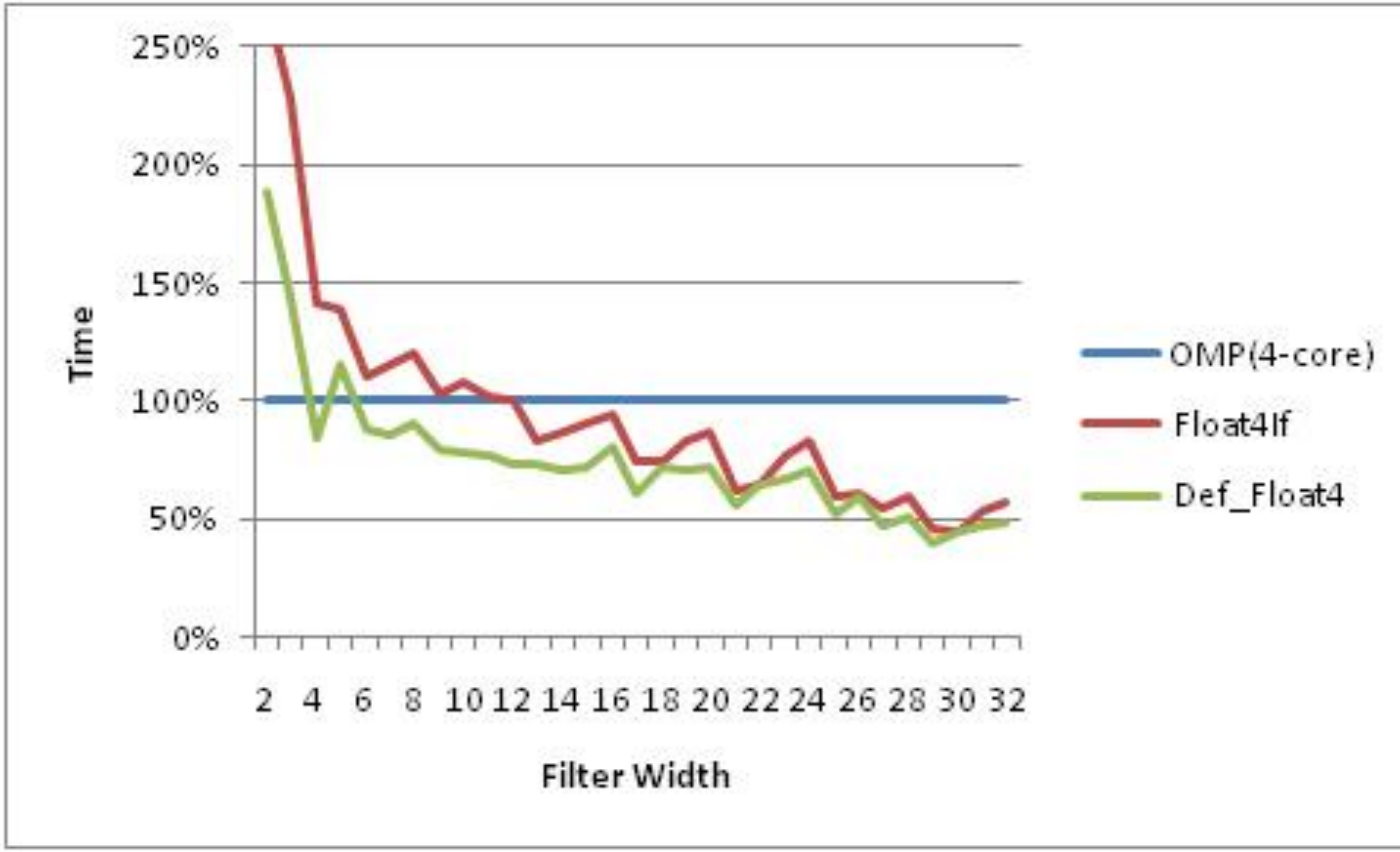


Instead of passing filterWidth as an argument to the kernel, we will define the value for FILTER_WIDTH when we build the OpenCL program object

OpenMP Comparison



OpenMP Comparison



OpenCL Sorting

Eric Bainville - June 2011

Parallel Selection Sort

```
__kernel void ParallelSelection(__global const data_t * in,__global
data_t * out)
{
  int i = get_global_id(0); // current thread
  int n = get_global_size(0); // input size
  data_t iData = in[i];
  uint iKey = keyValue(iData);
  // Compute position of in[i] in output
  int pos = 0;
  for (int j=0;j<n;j++)
  {
    uint jKey = keyValue(in[j]); // broadcasted
    // in[j] < in[i] ?
    bool smaller = (jKey < iKey) || (jKey == iKey && j < i);
    pos += (smaller)?1:0;
  }
  out[pos] = iData;
}
```

Parallel Selection Sort

- Very ineffective
- $2N+N^2$ accesses to global memory. Why?

- A.k.a. Parallel Rank Sort
  - Effective on multi-processor system with high-bandwidth memory

Parallel Selection Sort, blocks

```
__kernel void ParallelSelection_Blocks(__global const data_t *
in,__global data_t * out,__local uint * aux)
{
  int i = get_global_id(0); // current thread
  int n = get_global_size(0); // input size
  int wg = get_local_size(0); // workgroup size

  data_t iData = in[i]; // input record for current thread
  uint iKey = keyValue(iData); // input key for current thread
  int blockSize = BLOCK_FACTOR * wg; // block size
```

```
  // Compute position of iKey in output
  int pos = 0;
  // Loop on blocks of size BLOCKSIZE keys (BLOCKSIZE must divide N)
  for (int j=0;j<n;j+=blockSize)
  {
    // Load BLOCKSIZE keys using all threads (BLOCK_FACTOR values per thread)
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int index=get_local_id(0);index<blockSize;index+=wg)
      aux[index] = keyValue(in[j+index]);
    barrier(CLK_LOCAL_MEM_FENCE);

    // Loop on all values in AUX
    for (int index=0;index<blockSize;index++)
    {
      uint jKey = aux[index]; // broadcasted, local memory
      // in[j] < in[i] ?
      bool smaller = (jKey < iKey) || ( jKey == iKey && (j+index) < i );
      pos += (smaller)?1:0;
    }
  }
  out[pos] = iData;
}
```

Compare-and-Exchange Sorting

Fikret Ercal
Missouri University of Science and Technology

Parallel Rank Sort with $P=N^2$

- P processors, N data items
- Can you propose an $O(\log N)$ algorithm?

Odd-Even Transposition Sort

Time / Step	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	4	2	7	8	5	1	3	6
1	2	4	7	8	1	5	3	6
2	2	4	7	1	8	3	5	6
3	2	4	1	7	3	8	5	6
4	2	1	4	3	7	5	8	6
5	1	2	3	4	5	7	6	8
6	1	2	3	4	5	6	7	8
7	1	2	3	4	5	6	7	8

Parallel time complexity: $T_{par} = O(N)$ (for $P=N$)

Odd-Even Transposition Sort (N>>P)

Each PE gets N/P numbers. First, PEs sort N/P locally, then they run odd-even trans. algorithm each time doing a merge-split for 2N/P numbers.

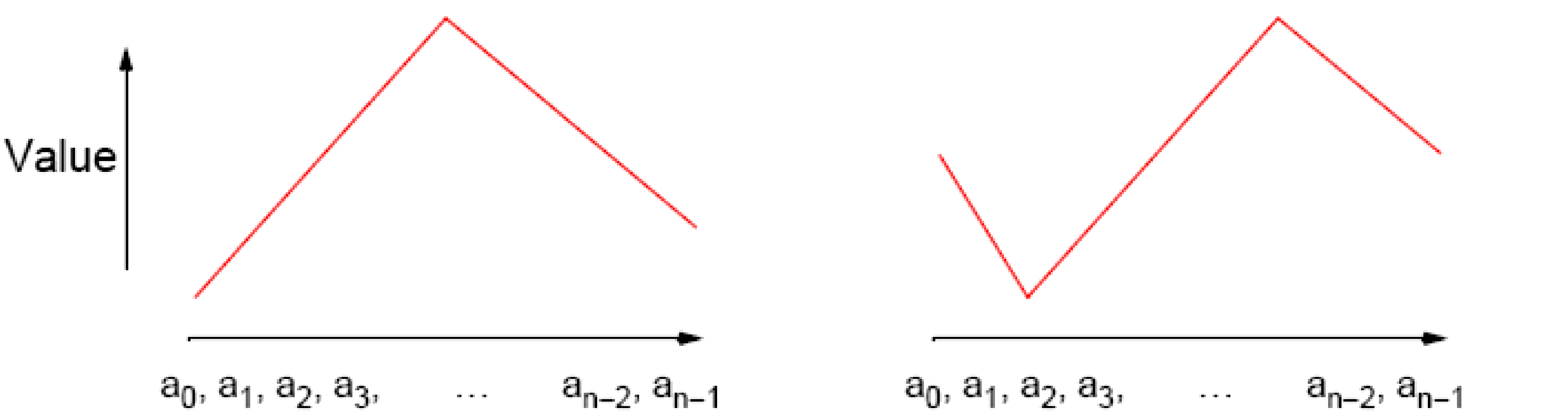
	P_0	P_1	P_2	P_3
	13 7 12	8 5 4	6 1 3	9 2 10
Local sort				
	7 12 13	4 5 8	1 3 6	2 9 10
O-E				
	4 5 7	8 12 13	1 2 3	6 9 10
E-O				
	4 5 7	1 2 3	8 12 13	6 9 10
O-E				
	1 2 3	4 5 7	6 8 9	10 12 13
E-O				
SORTED:	1 2 3	4 5 6	7 8 9	10 12 13

Time complexity: T_{par} = (Local Sort) + (p merge-splits) +(p exchanges)

$$T_{par} = (n/p)\log(n/p) + p^*(n/p) + p^*(n/p) = (n/p)\log(n/p) + 2n$$

Bitonic Mergesort

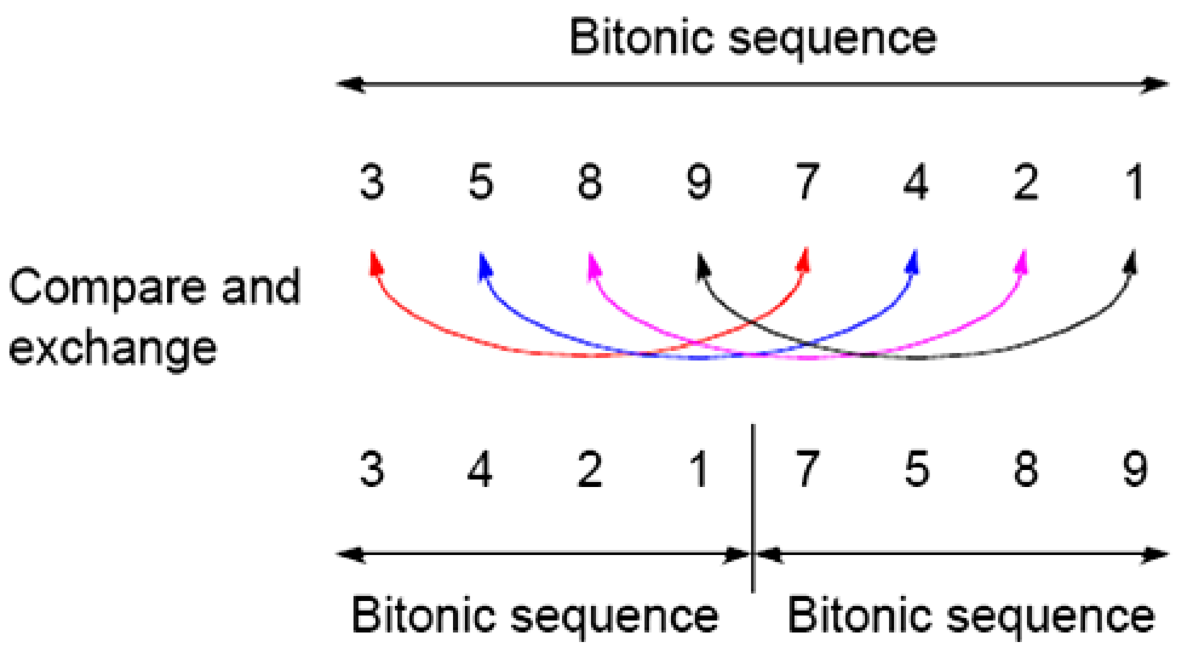
A bitonic sequence is defined as a list with no more than one LOCAL MAXIMUM and no more than one LOCAL MINIMUM. (Endpoints must be considered - wraparound)



(a) Single maximum (b) Single maximum and single minimum

Binary Split

1. Divide the bitonic list into two equal halves.
2. Compare-Exchange each item on the first half with the corresponding item in the second half.



Result:

Two bitonic sequences where the numbers in one sequence are all less than the numbers in the other sequence.

Repeated Application of Binary Split

Bitonic list:

24 20 15 9 4 2 5 8 | 10 11 12 13 22 30 32 45

Result after Binary-split:

10 11 12 9 4 2 5 8 | 24 20 15 13 22 30 32 45

If you keep applying the BINARY-SPLIT to each half repeatedly, you will get a SORTED LIST !

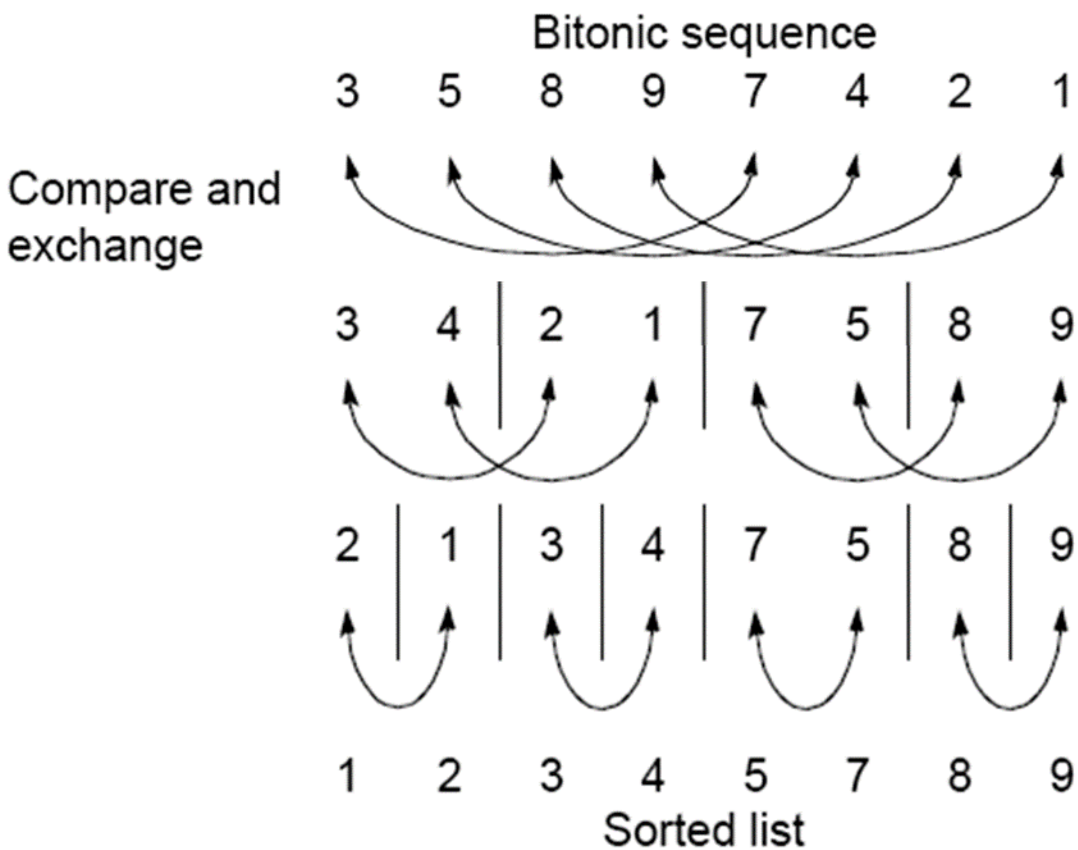
```
10   11   12   9 .  4   2    5   8 | 24  20   15   13 .  22  30   32  45
4     2 .  5   8   10  11 . 12   9 |  22  20 . 15   13    24  30 . 32  45
4  .  2    5 . 8   10 .  9   12 .11    15 . 13   22 . 20   24 . 30  32 . 45
2     4    5   8    9  10    11  12    13   15   20   22   24   30  32   45
```

Q: How many parallel steps does it take to sort ?

A: log n

Sorting a Bitonic Sequence

- Compare-and-exchange moves smaller numbers of each pair to left and larger numbers of pair to right.
- Given a bitonic sequence, recursively performing 'binary split' will sort the list.



Sorting an Arbitrary Sequence

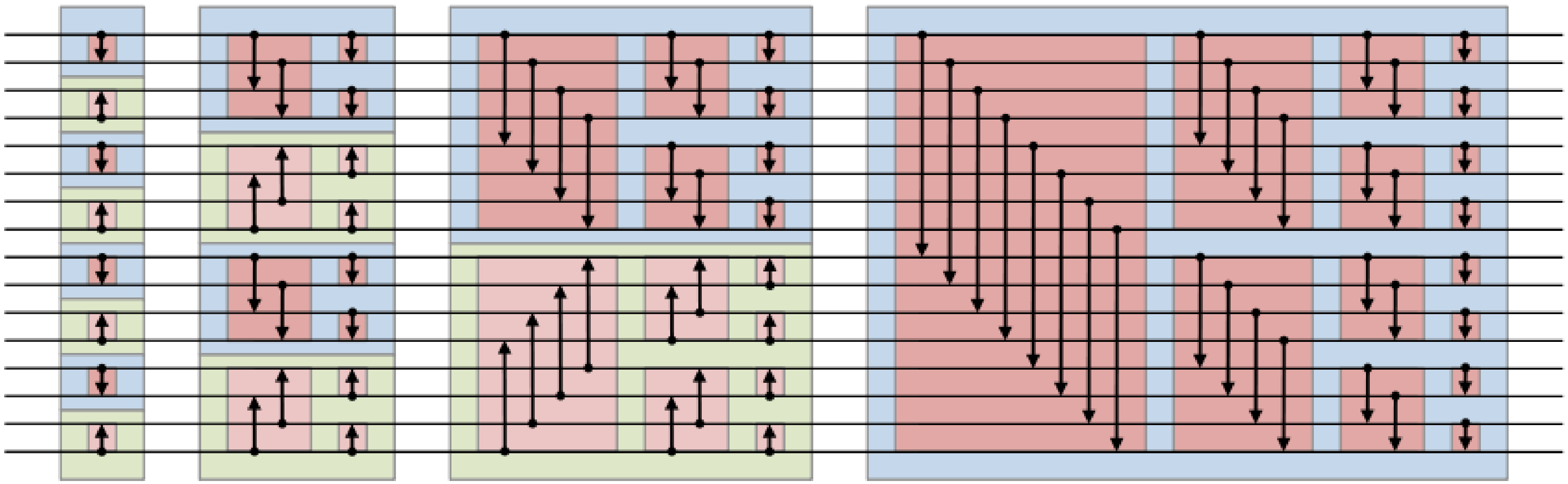
- To sort an unordered sequence, sequences are merged into larger bitonic sequences, starting with pairs of adjacent numbers.

- By a compare-and-exchange operation, pairs of adjacent numbers formed into increasing sequences and decreasing sequences. Pairs form a bitonic sequence of twice the size of each original sequences.

- By repeating this process, bitonic sequences of larger and larger lengths obtained.

- In the final step, a single bitonic sequence sorted into a single increasing sequence.

Bitonic Mergesort



- Whenever two numbers reach the two ends of an arrow, they are compared to ensure that the arrow points toward the larger number.
- If they are out of order, they are swapped.

Source: Wikipedia

Python Example

```
def bitonic_sort(up, x):
    if len(x) <= 1:
        return x
    else:
        first = bitonic_sort(True, x[:len(x) / 2])
        second = bitonic_sort(False, x[len(x) / 2:])
        return bitonic_merge(up, first + second)

def bitonic_merge(up, x):
    # assume input x is bitonic, and sorted list is returned
    if len(x) == 1:
        return x
    else:
        bitonic_compare(up, x)
        first = bitonic_merge(up, x[:len(x) / 2])
        second = bitonic_merge(up, x[len(x) / 2:])
        return first + second

def bitonic_compare(up, x):
    dist = len(x) / 2
    for i in range(dist):
        if (x[i] > x[i + dist]) == up:
            x[i], x[i + dist] = x[i + dist], x[i] #swap
```

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

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

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;
```

Uses whatever mapping to threads and blocks the compiler chooses

```
#pragma acc kernels loop
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 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 parallel

```
#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];
}
```

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

Tips and Tricks

- Use `restrict` keyword

```
float *restrict ptr
```

Meaning: “for the lifetime of ptr, only it or a value directly derived from it (such as ptr + 1) will be used to access the object to which it points”

- Limits the effects of pointer aliasing
- Compilers often require restrict to determine independence (true for OpenACC, OpenMP, and vectorization)
- Otherwise the compiler can’t parallelize loops that access ptr
- Note: if programmer violates the declaration, behavior is undefined

Tips and Tricks

- Nested loops are best for parallelization
  - Large loop counts (1000s) needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: use restrict keyword in C
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
- Use subscripted arrays, rather than pointer-indexed arrays
- Use contiguous memory for multi-dimensional arrays