Research master project at LIT
1 September 2011 – 1 September 2013

Accelerating sequential computer vision algorithms using commodity parallel hardware

NIOC 4 April 2013

Jaap van de Loosdrecht

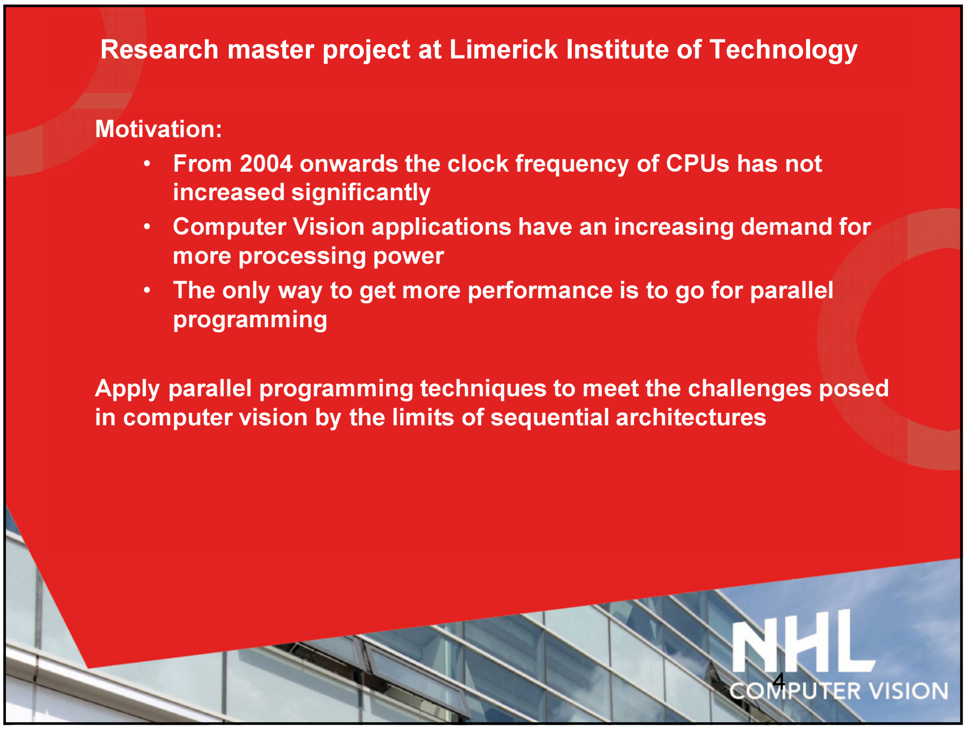
NHL Centre of Expertise in Computer Vision
Van de Loosdrecht Machine Vision BV

Overview

- Background and motivation
- CPU versus GPU architecture
- Choice of the standards for multi-core programming
    - OpenMP
    - OpenCL
    - Benchmark examples
    - Evaluation and recommendations
- Future work
- Conclusions

Background

- NHL Centre of Expertise in Computer Vision
  - Founded 1996
  - 4,5 FTE
  - 170 industrial projects, > 3.000.000 euro
  - Course CV (10 Universities of Professional Education)
- Van de Loosdrecht Machine Vision BV
  - Development started 1993, founded 2001
  - VisionLab: development environment for
    - Computer vision
    - Pattern matching
    - Neural networks
    - Genetic algorithms
  - Portable library, > 100.000 lines of ANSI C++
    Windows, Linux and Android
    x86, x64, ARM and PowerPC

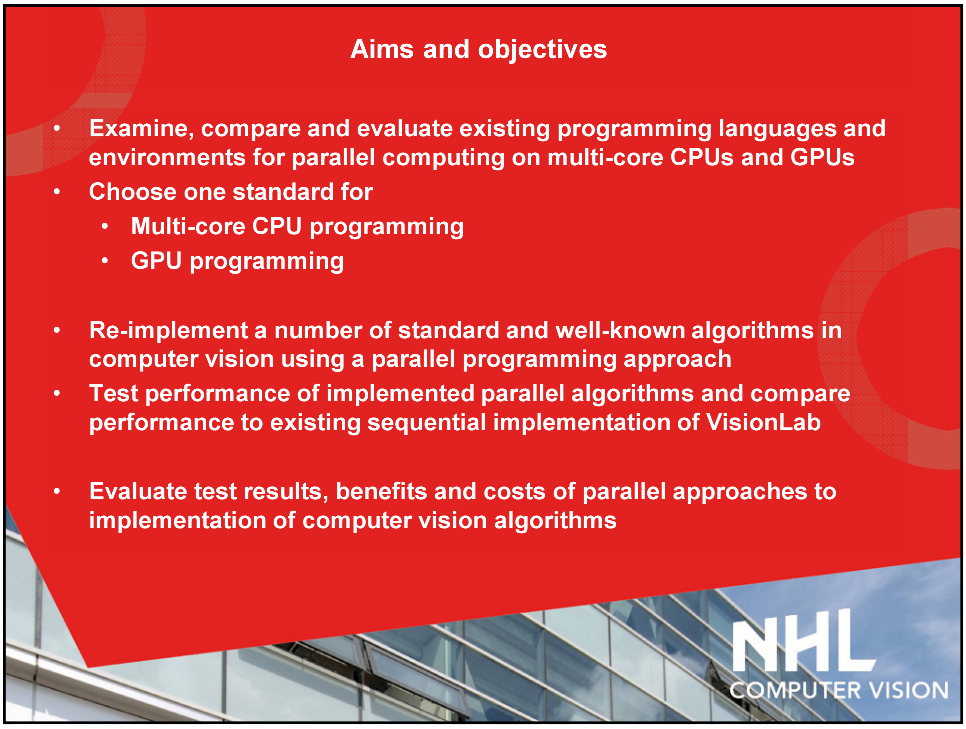


Research master project at Limerick Institute of Technology

Motivation:

- From 2004 onwards the clock frequency of CPUs has not increased significantly
- Computer Vision applications have an increasing demand for more processing power
- The only way to get more performance is to go for parallel programming

Apply parallel programming techniques to meet the challenges posed in computer vision by the limits of sequential architectures



Aims and objectives

- Examine, compare and evaluate existing programming languages and environments for parallel computing on multi-core CPUs and GPUs
- Choose one standard for
  - Multi-core CPU programming
  - GPU programming

- Re-implement a number of standard and well-known algorithms in computer vision using a parallel programming approach
- Test performance of implemented parallel algorithms and compare performance to existing sequential implementation of VisionLab

- Evaluate test results, benefits and costs of parallel approaches to implementation of computer vision algorithms

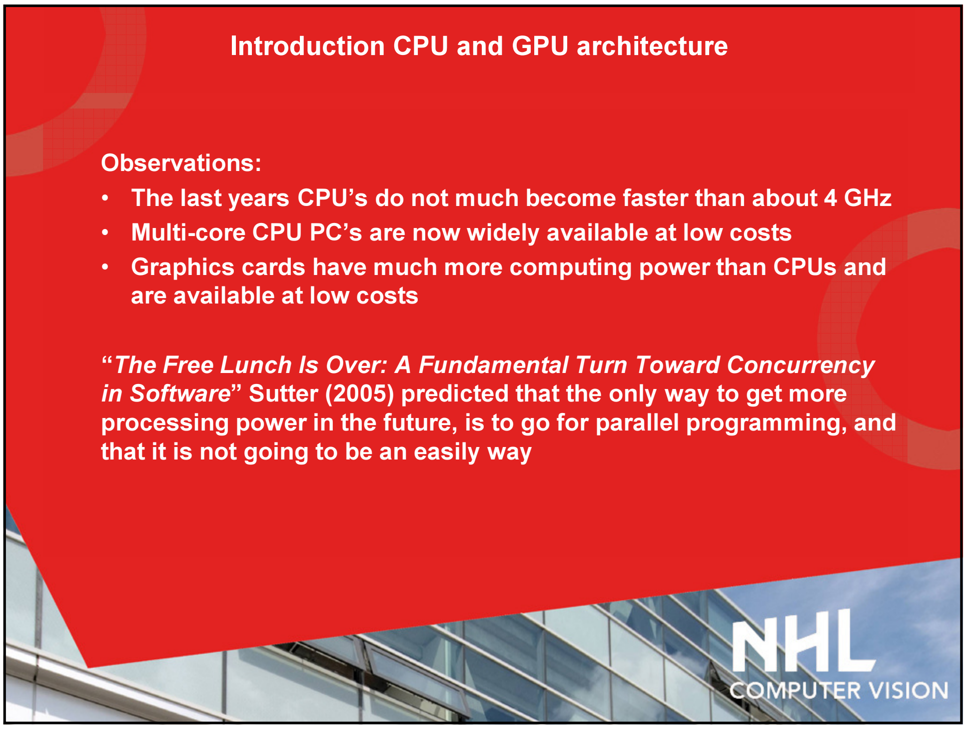
Related research

Other research projects:

- Quest for one domain specific algorithm to compare the best sequential with best parallel implementation on a specific hardware
- Framework for auto parallelisation or vectorization
  In research, not yet generic applicable

This project is distinctive:

- Investigate how to speedup a whole library by parallelizing the algorithms in an economical way and execute them on multiple platforms
  - 100.000 lines of ANSI C++ code
  - Generic library
  - Portability and vendor independency
  - Variance in execution times
  - Run time prediction if parallelization is beneficial

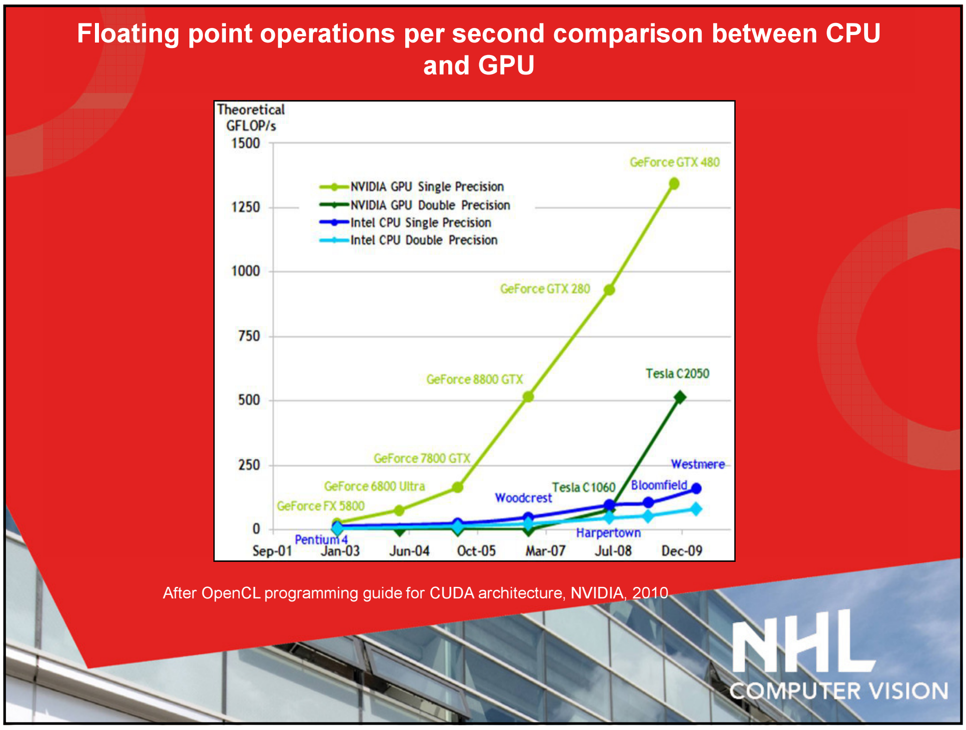


Introduction CPU and GPU architecture

Observations:

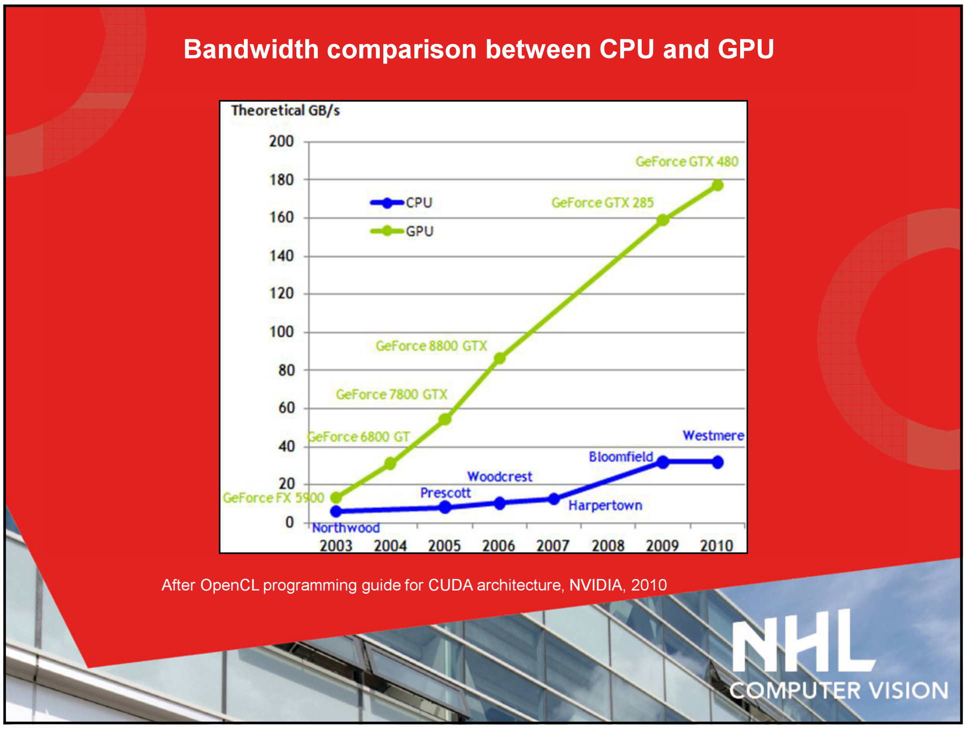
- The last years CPU’s do not much become faster than about 4 GHz
- Multi-core CPU PC’s are now widely available at low costs
- Graphics cards have much more computing power than CPUs and are available at low costs

“*The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*” Sutter (2005) predicted that the only way to get more processing power in the future, is to go for parallel programming, and that it is not going to be an easily way



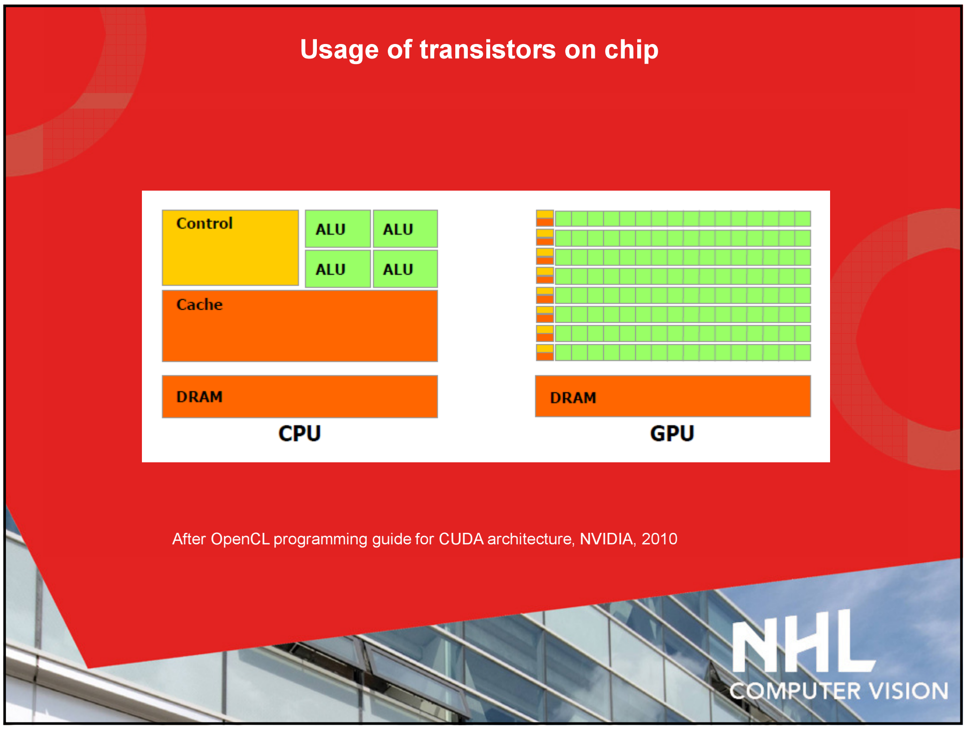
Floating point operations per second comparison between CPU and GPU

After OpenCL programming guide for CUDA architecture, NVIDIA, 2010



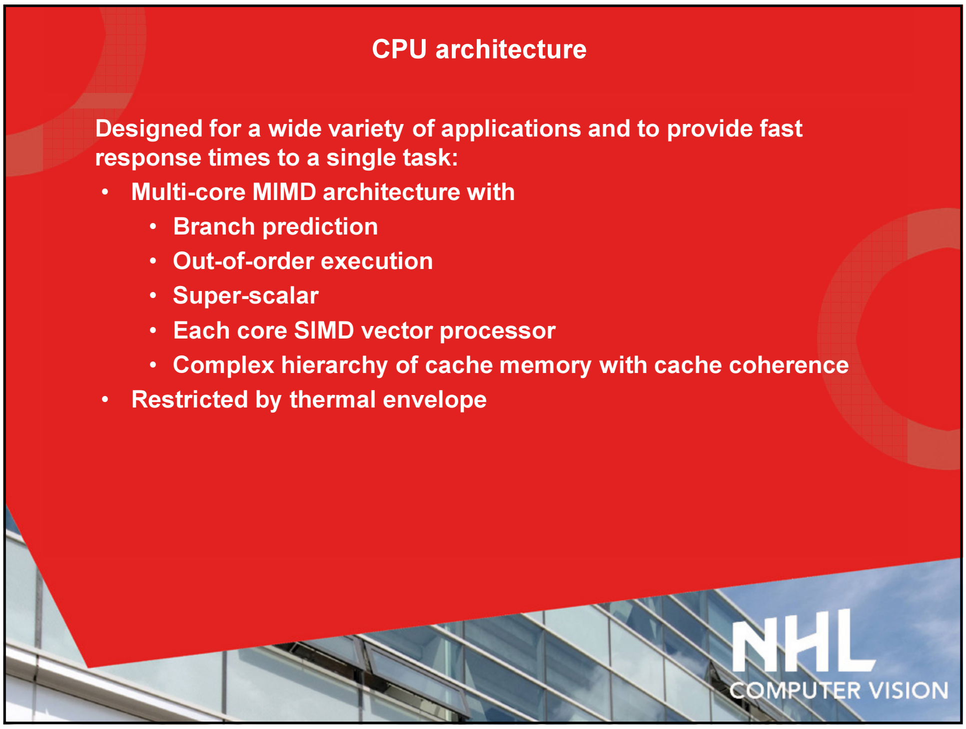
Bandwidth comparison between CPU and GPU

After OpenCL programming guide for CUDA architecture, NVIDIA, 2010



Usage of transistors on chip

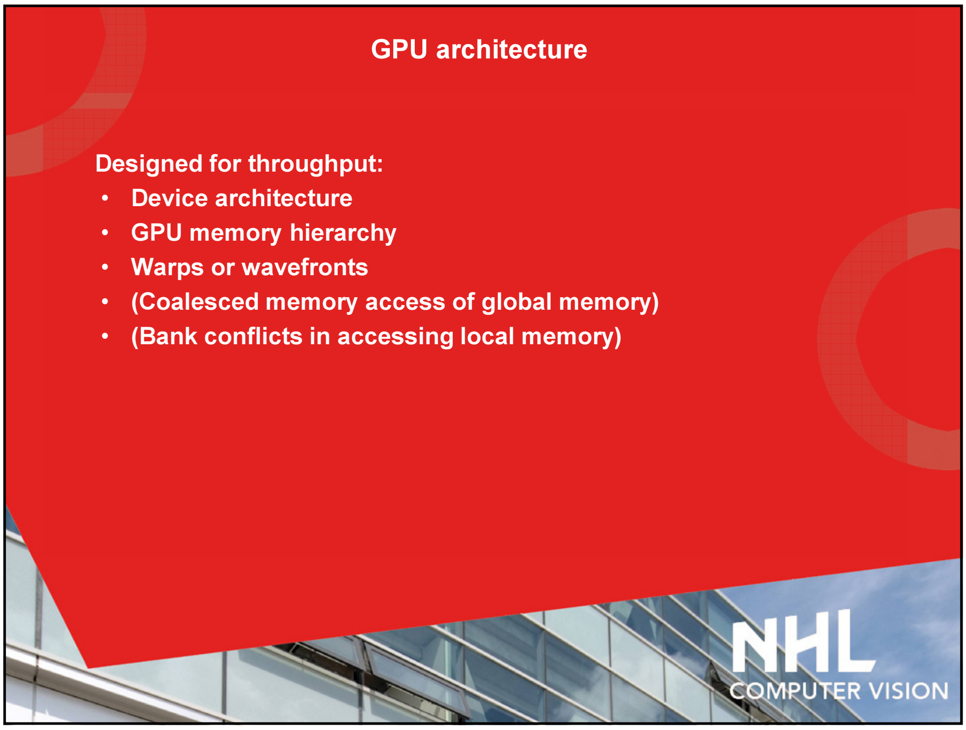
After OpenCL programming guide for CUDA architecture, NVIDIA, 2010



CPU architecture

Designed for a wide variety of applications and to provide fast response times to a single task:

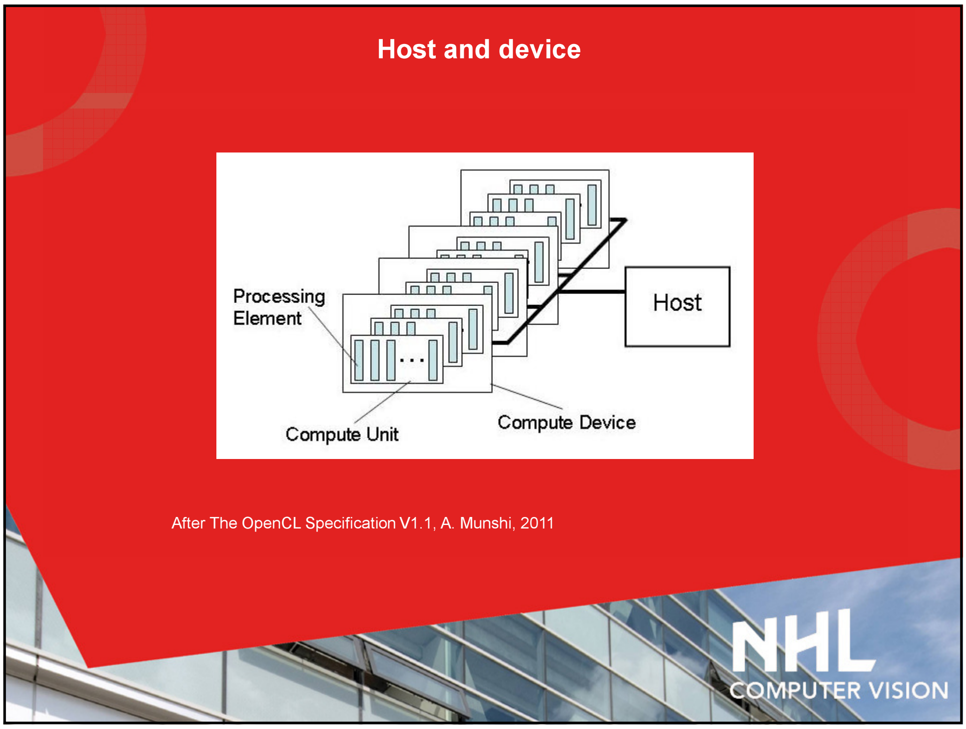
- Multi-core MIMD architecture with
  - Branch prediction
  - Out-of-order execution
  - Super-scalar
  - Each core SIMD vector processor
  - Complex hierarchy of cache memory with cache coherence
- Restricted by thermal envelope



GPU architecture

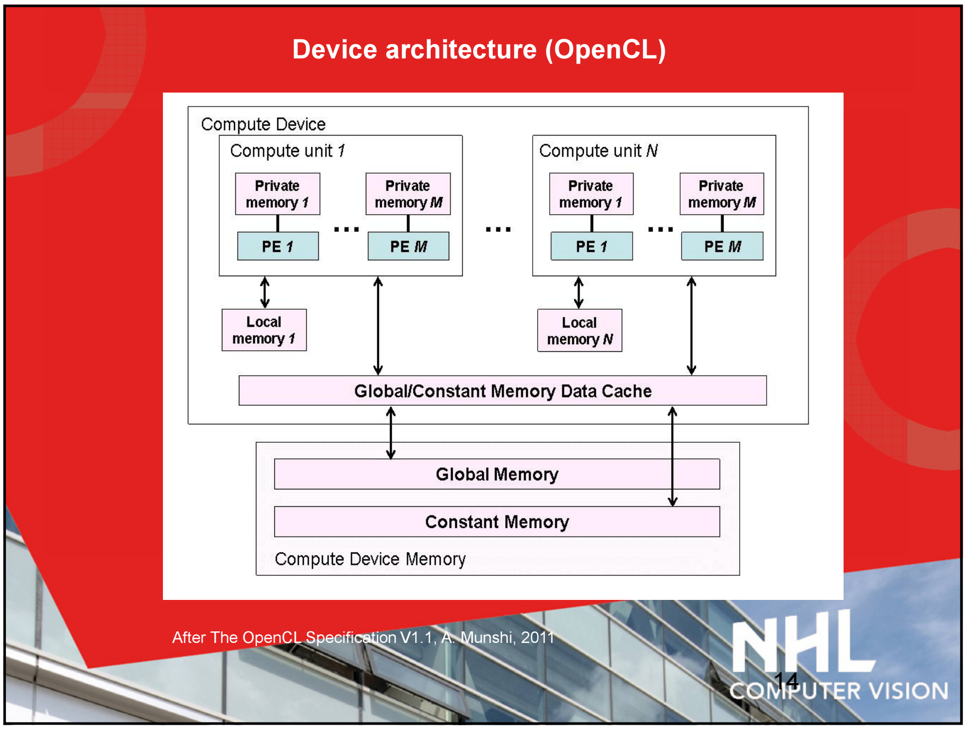
Designed for throughput:

- Device architecture
- GPU memory hierarchy
- Warps or wavefronts
- (Coalesced memory access of global memory)
- (Bank conflicts in accessing local memory)



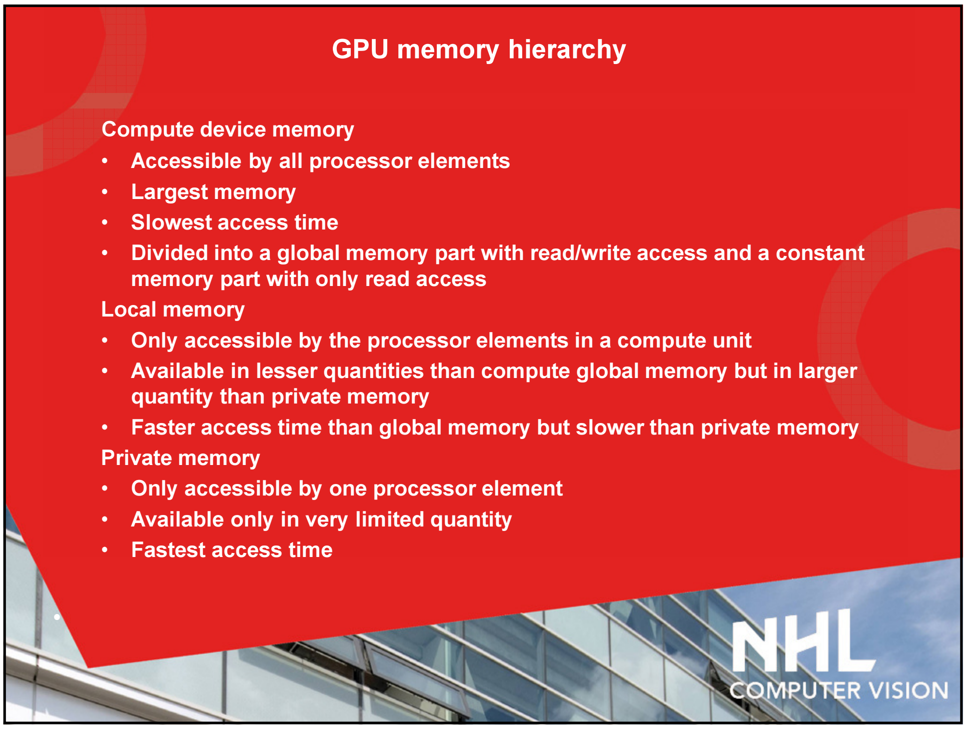
Host and device

After The OpenCL Specification V1.1, A. Munshi, 2011



Device architecture (OpenCL)

After The OpenCL Specification V1.1, A. Munshi, 2011



GPU memory hierarchy

Compute device memory

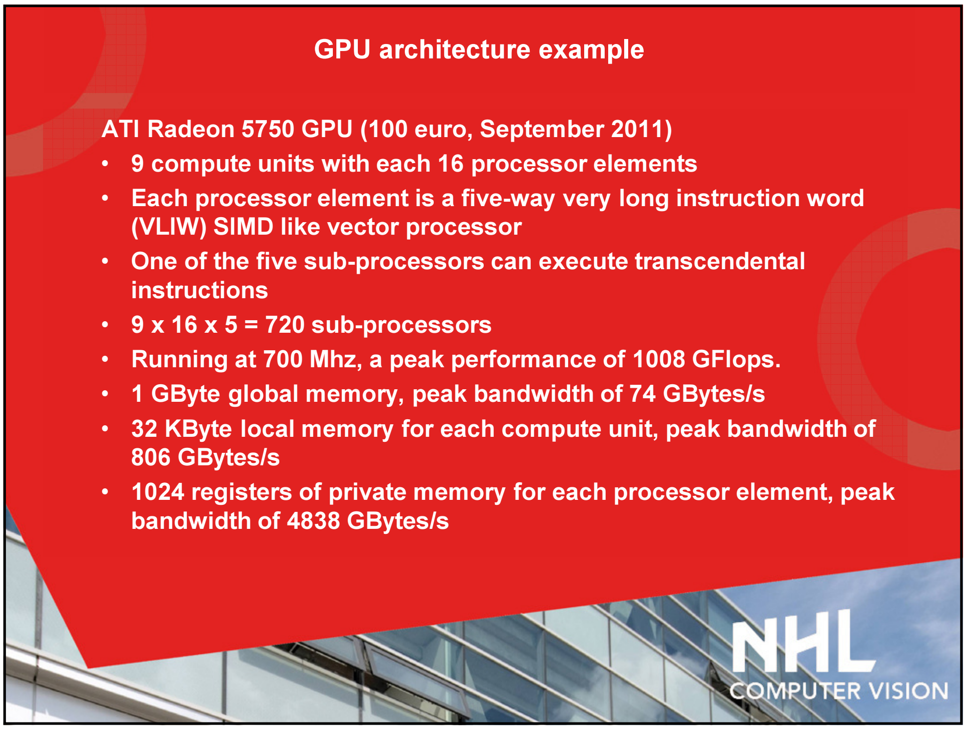
- Accessible by all processor elements
- Largest memory
- Slowest access time
- Divided into a global memory part with read/write access and a constant memory part with only read access

Local memory

- Only accessible by the processor elements in a compute unit
- Available in lesser quantities than compute global memory but in larger quantity than private memory
- Faster access time than global memory but slower than private memory

Private memory

- Only accessible by one processor element
- Available only in very limited quantity
- Fastest access time



GPU architecture example

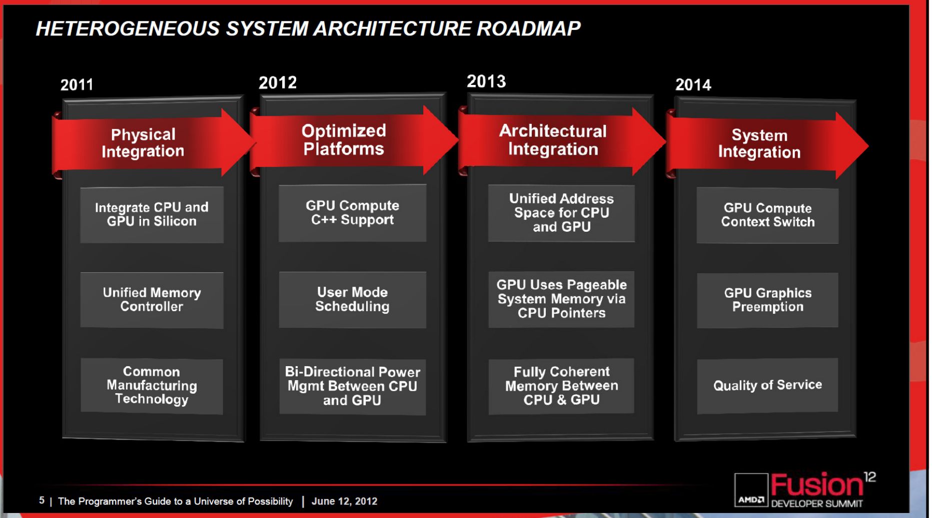
ATI Radeon 5750 GPU (100 euro, September 2011)

- 9 compute units with each 16 processor elements
- Each processor element is a five-way very long instruction word (VLIW) SIMD like vector processor
- One of the five sub-processors can execute transcendental instructions
- 9 x 16 x 5 = 720 sub-processors
- Running at 700 Mhz, a peak performance of 1008 GFlops.
- 1 GByte global memory, peak bandwidth of 74 GBytes/s
- 32 KByte local memory for each compute unit, peak bandwidth of 806 GBytes/s
- 1024 registers of private memory for each processor element, peak bandwidth of 4838 GBytes/s

Warps or wavefronts

- All work-items in a warp execute the same instruction in parallel on all cores of a compute unit (SIMT)
- Typical size for a warp is 16, 32 or 64 work-items
- Branch diversion
- Zero-overhead warp scheduling
- Occupancy rate
  Hide long global memory latency

HSA Roadmap



HETEROGENEOUS SYSTEM ARCHITECTURE ROADMAP

2011

Physical Integration

- Integrate CPU and GPU in Silicon
- Unified Memory Controller
- Common Manufacturing Technology

2012

Optimized Platforms

- GPU Compute C++ Support
- User Mode Scheduling
- Bi-Directional Power Mgmt Between CPU and GPU

2013

Architectural Integration

- Unified Address Space for CPU and GPU
- GPU Uses Pageable System Memory via CPU Pointers
- Fully Coherent Memory Between CPU & GPU

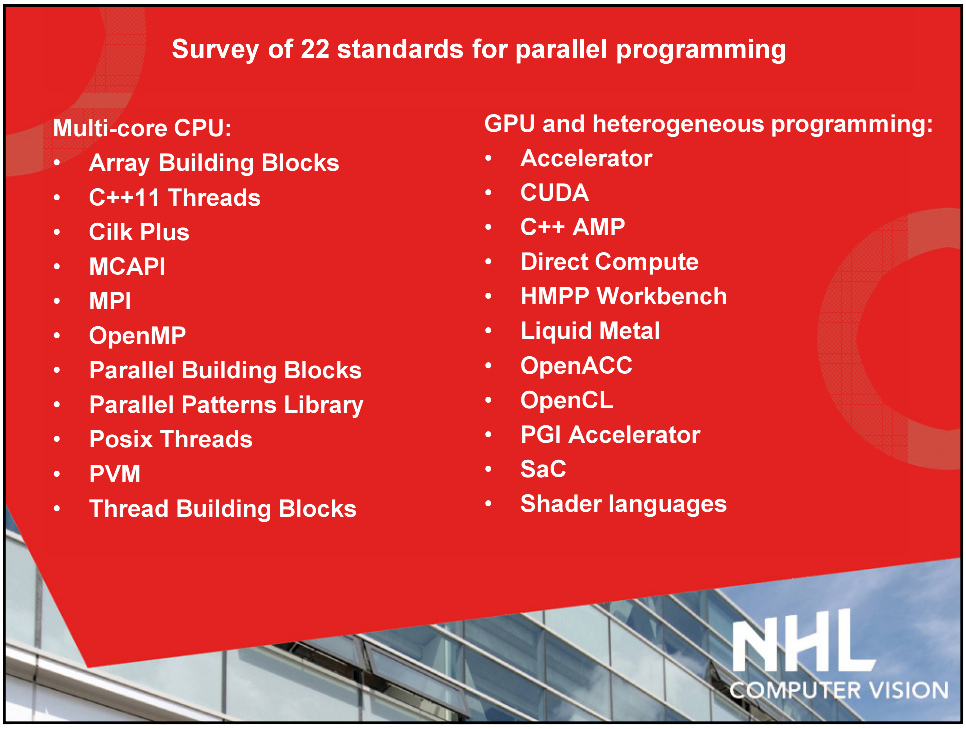
2014

System Integration

- GPU Compute Context Switch
- GPU Graphics Preemption
- Quality of Service

5 | The Programmer's Guide to a Universe of Possibility | June 12, 2012

After P. Rogers, AMD 2012



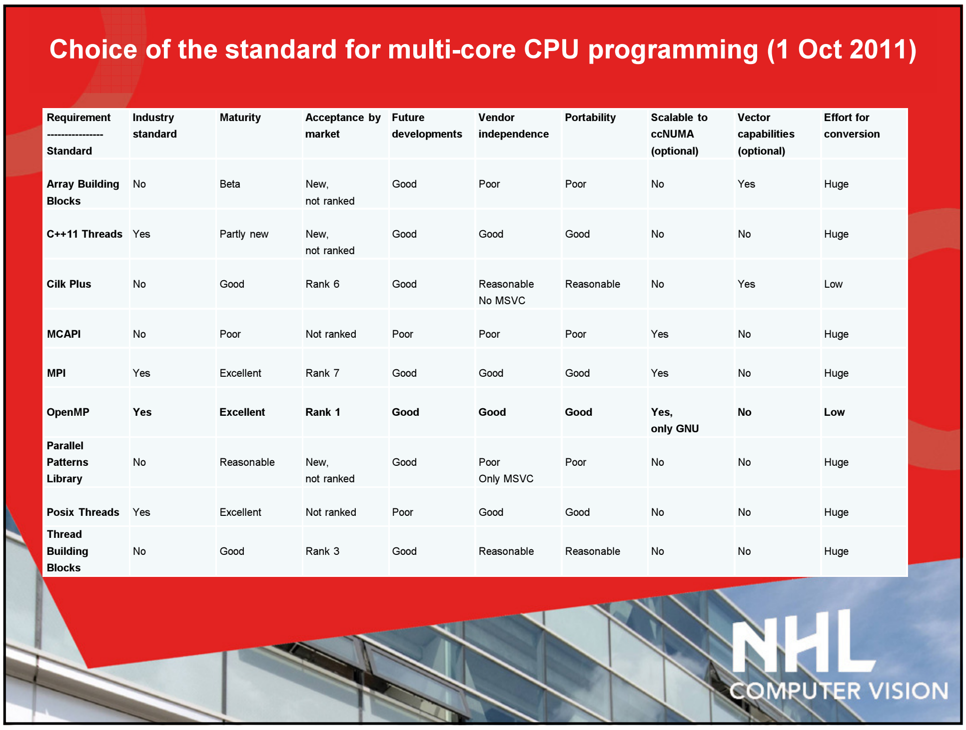
Survey of 22 standards for parallel programming

Multi-core CPU:

- Array Building Blocks
- C++11 Threads
- Cilk Plus
- MCAPI
- MPI
- OpenMP
- Parallel Building Blocks
- Parallel Patterns Library
- Posix Threads
- PVM
- Thread Building Blocks

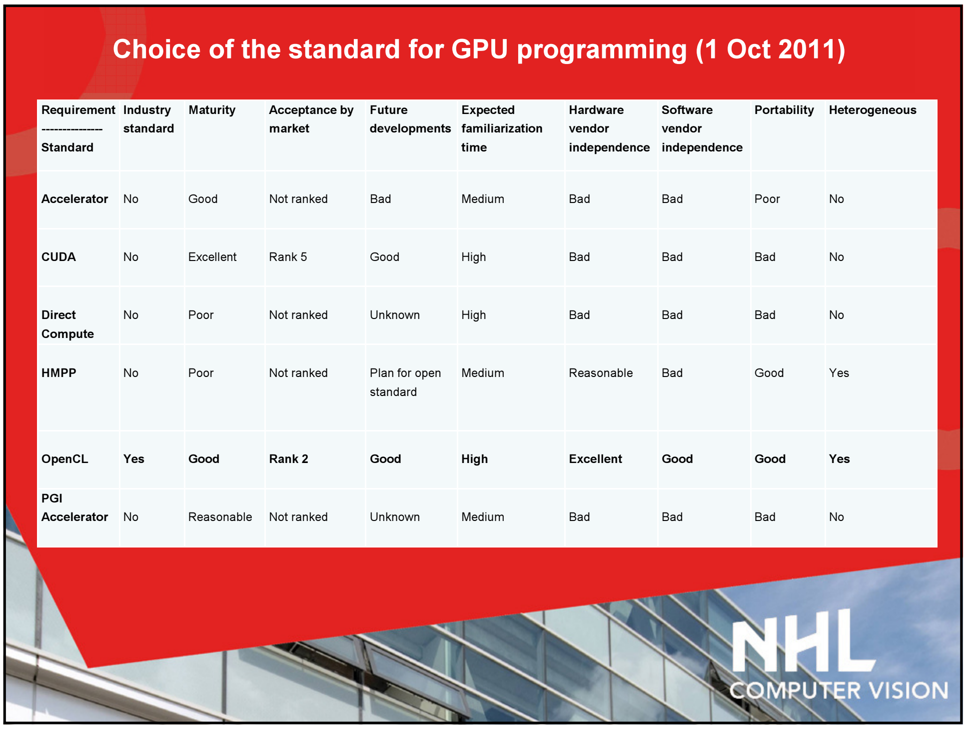
GPU and heterogeneous programming:

- Accelerator
- CUDA
- C++ AMP
- Direct Compute
- HMPP Workbench
- Liquid Metal
- OpenACC
- OpenCL
- PGI Accelerator
- SaC
- Shader languages



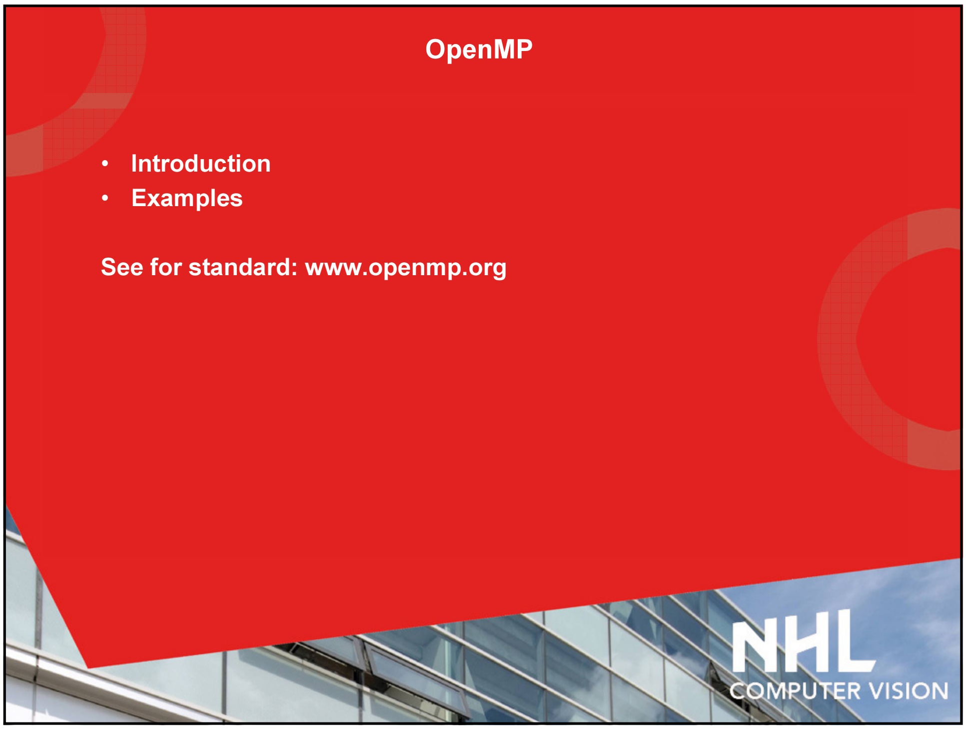
Choice of the standard for multi-core CPU programming (1 Oct 2011)

Requirement --------------- Standard	Industry standard	Maturity	Acceptance by market	Future developments	Vendor independence	Portability	Scalable to ccNUMA (optional)	Vector capabilities (optional)	Effort for conversion
Array Building Blocks	No	Beta	New, not ranked	Good	Poor	Poor	No	Yes	Huge
C++11 Threads	Yes	Partly new	New, not ranked	Good	Good	Good	No	No	Huge
Cilk Plus	No	Good	Rank 6	Good	Reasonable No MSVC	Reasonable	No	Yes	Low
MCAPI	No	Poor	Not ranked	Poor	Poor	Poor	Yes	No	Huge
MPI	Yes	Excellent	Rank 7	Good	Good	Good	Yes	No	Huge
OpenMP	**Yes**	**Excellent**	**Rank 1**	**Good**	**Good**	**Good**	**Yes, only GNU**	**No**	**Low**
Parallel Patterns Library	No	Reasonable	New, not ranked	Good	Poor Only MSVC	Poor	No	No	Huge
Posix Threads	Yes	Excellent	Not ranked	Poor	Good	Good	No	No	Huge
Thread Building Blocks	No	Good	Rank 3	Good	Reasonable	Reasonable	No	No	Huge



Choice of the standard for GPU programming (1 Oct 2011)

Requirement **--------------** **Standard**	**Industry standard**	**Maturity**	**Acceptance by market**	**Future developments**	**Expected familiarization time**	**Hardware vendor independence**	**Software vendor independence**	**Portability**	**Heterogeneous**
Accelerator	No	Good	Not ranked	Bad	Medium	Bad	Bad	Poor	No
CUDA	No	Excellent	Rank 5	Good	High	Bad	Bad	Bad	No
Direct Compute	No	Poor	Not ranked	Unknown	High	Bad	Bad	Bad	No
HMPP	No	Poor	Not ranked	Plan for open standard	Medium	Reasonable	Bad	Good	Yes
OpenCL	**Yes**	**Good**	**Rank 2**	**Good**	**High**	**Excellent**	**Good**	**Good**	**Yes**
PGI Accelerator	No	Reasonable	Not ranked	Unknown	Medium	Bad	Bad	Bad	No



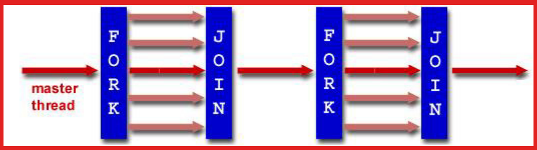
OpenMP

- Introduction
- Examples

See for standard: www.openmp.org

OpenMP introduction

- An API that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran
- Supports both data parallel and task parallel multiprocessing
- Fork-join programming model



After Introduction to Parallel Computing, Barney, 2011

NHL COMPUTER VISION

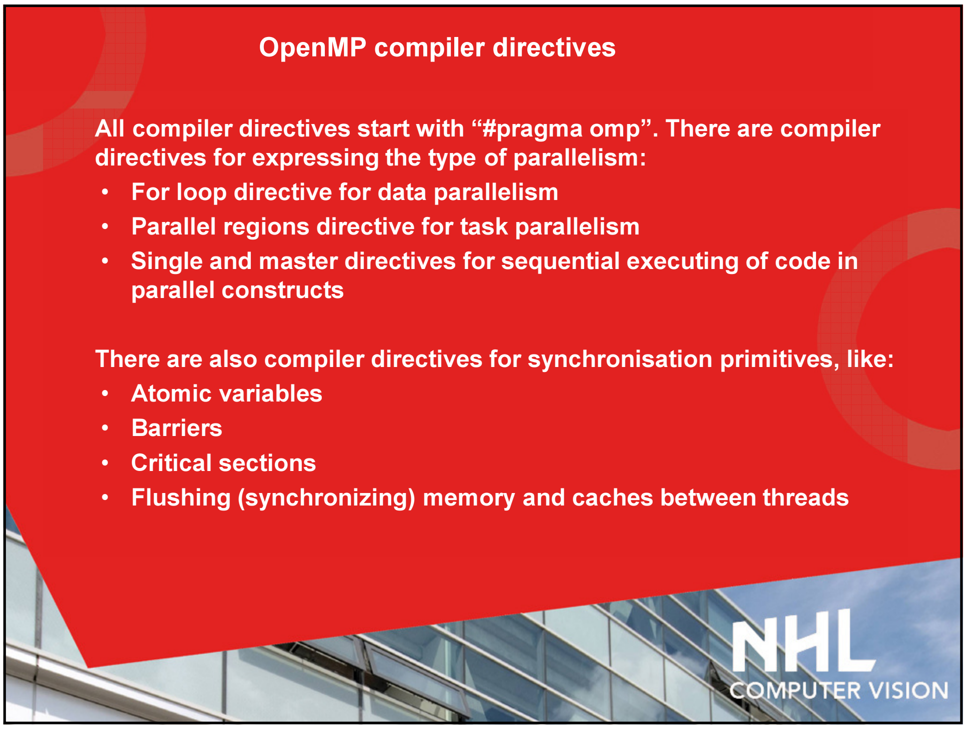
OpenMP example adding two vectors

```
const int SIZE = 1000;
double a[SIZE], b[SIZE], c[SIZE];
// code for initialising array b and c
#pragma omp for
for (int j = 0; j < SIZE; j++) {
    a[j] = b[j] + c[j];
} // for j
```

Assuming CPU has four cores, at executing time the next events will happen:

- The master thread forks a team of three threads
- All four threads will execute in parallel one quarter of the for loop. The first thread will execute the for loop for 0 <= j < 250, the second thread will execute the for loop for 250 <= j < 500, etc
- When all threads have completed their work the threads will join

NHL COMPUTER VISION



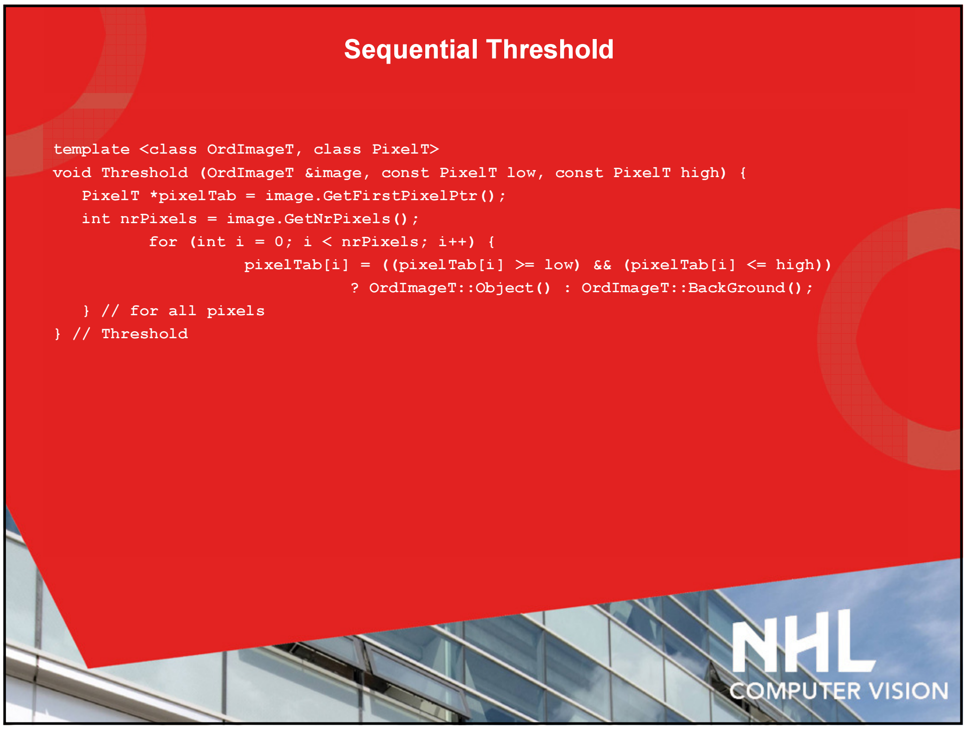
OpenMP compiler directives

All compiler directives start with “#pragma omp”. There are compiler directives for expressing the type of parallelism:

- For loop directive for data parallelism
- Parallel regions directive for task parallelism
- Single and master directives for sequential executing of code in parallel constructs

There are also compiler directives for synchronisation primitives, like:

- Atomic variables
- Barriers
- Critical sections
- Flushing (synchronizing) memory and caches between threads



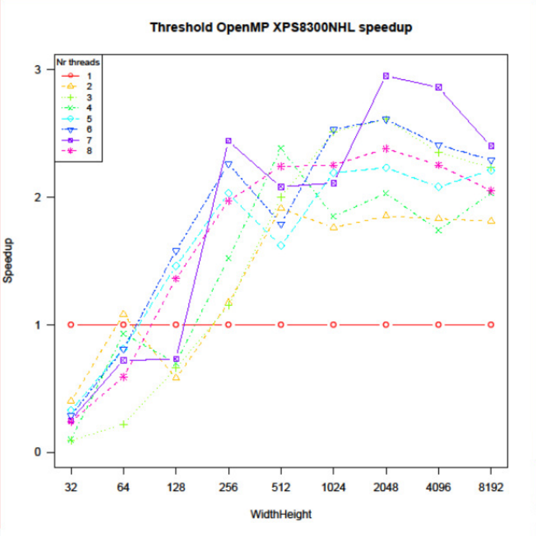
Sequential Threshold

```
template <class OrdImageT, class PixelT>
void Threshold (OrdImageT &image, const PixelT low, const PixelT high) {
   PixelT *pixelTab = image.GetFirstPixelPtr();
   int nrPixels = image.GetNrPixels();
          for (int i = 0; i < nrPixels; i++) {
                    pixelTab[i] = ((pixelTab[i] >= low) && (pixelTab[i] <= high))
                               ? OrdImageT::Object() : OrdImageT::BackGround();
   } // for all pixels
} // Threshold
```

OpenMP Threshold

```
template <class OrdImageT, class PixelT>
void Threshold (OrdImageT &image, const PixelT low, const PixelT high) {
   PixelT *pixelTab = image.GetFirstPixelPtr();
   int nrPixels = image.GetNrPixels();
#pragma omp parallel for
         for (int i = 0; i < nrPixels; i++) {
                   pixelTab[i] = ((pixelTab[i] >= low) && (pixelTab[i] <= high))
                              ? OrdImageT::Object() : OrdImageT::BackGround();
   } // for all pixels
} // Threshold
```

OpenMP Threshold speedup graph on i7 2600 (4 cores)



Sequential Histogram

```
template <class IntImageT>
void Histogram0 (const IntImageT &image, const int hisSize, int *his) {
   typedef typename IntImageT::PixelType PixelT;
   memset(his, 0, hisSize  * sizeof(int));
   PixelT *pixelTab = image.GetFirstPixelPtr();
   const int nrPixels = image.GetNrPixels();
   for (int i = 0; i < nrPixels; i++) {
      his[pixelTab[i]]++;
   } // for i
} // Histogram0
```

OpenMP Histogram

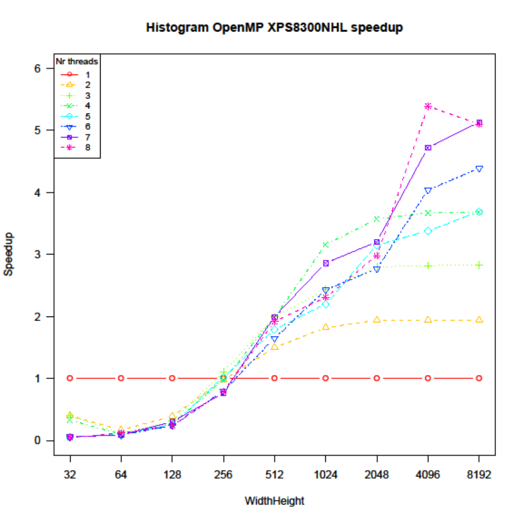
Pseudo code:

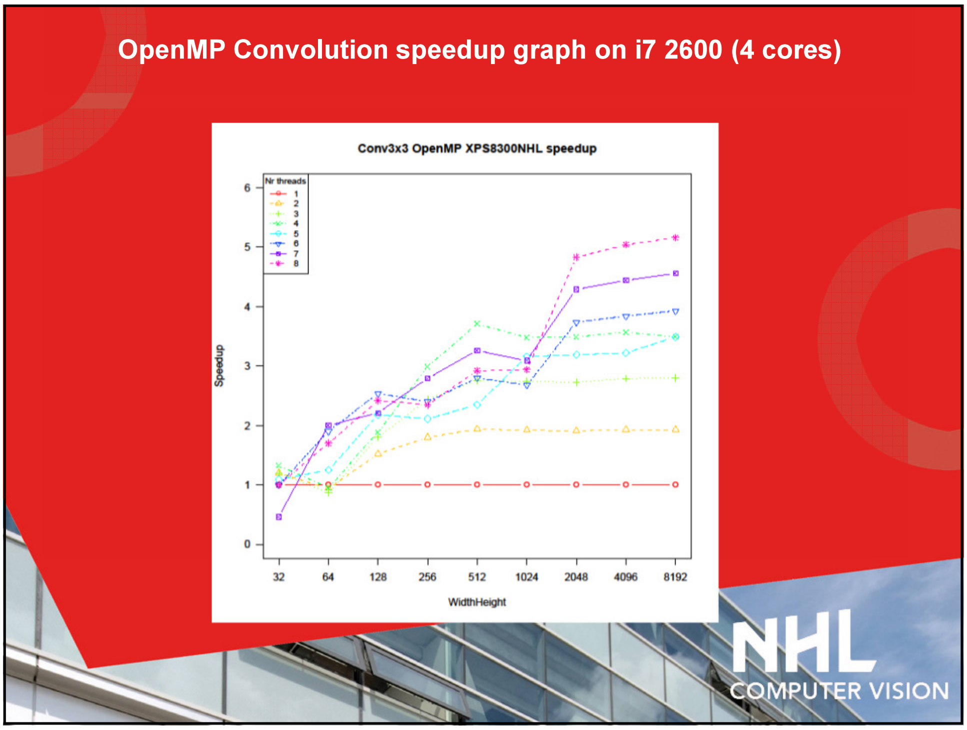
```
• Clear global histogram
• Split image in N parts and do in parallel for each part
    • Create and clear local histogram
    • Calculate local histogram
• Add all local histograms to global histogram
```

OpenMP Histogram

```
template <class IntImageT>
void Histogram0 (const IntImageT &image, const int hisSize, int *his) {
   typedef typename IntImageT::PixelType PixelT;
   memset(his, 0, hisSize * sizeof(int));
   PixelT *pixelTab = image.GetFirstPixelPtr();
   const int nrPixels = image.GetNrPixels();
#pragma omp parallel
   {
      int *localHis = new int[hisSize];
      memset(localHis, 0, hisSize * sizeof(int));
#pragma omp for nowait
      for (int i = 0; i < nrPixels; i++) {
         localHis[pixelTab[i]]++;
      } // for i
#pragma omp critical (CalcHistogram0)
      {
         for (int h = 0; h < hisSize; h++) {
            his[h] += localHis[h];
         } // for h
      } // omp critical
      delete [] localHis;
   } // omp parallel
} // Histogram0
```

OpenMP Histogram speedup graph on i7 2600 (4 cores)





OpenMP Convolution speedup graph on i7 2600 (4 cores)

OpenCL

- Kernel language
- Host API
- Examples

See for standard: www.khronos.org/opencl

OpenCL kernel language

Subset of ISO C99 with extensions

- No function pointers, recursion, bit fields, variable-length arrays and standard C99 header files
- Extensions: vector types, synchronization, functions to work with work-items/groups, etc

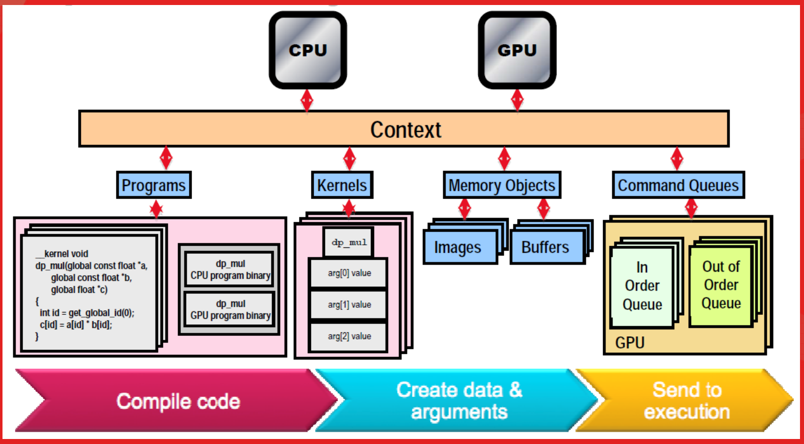
Proposal for OpenCL Static C++ Kernel Language Extension
Introduces C++ like features such as classes and templates

Kernel for adding of two vectors

```
kernel void VecAdd (global int* c, global int* a, global int* b) {
   unsigned int n = get_global_id(0);
   c[n] = a[n] + b[n];
}
```

NHL COMPUTER VISION

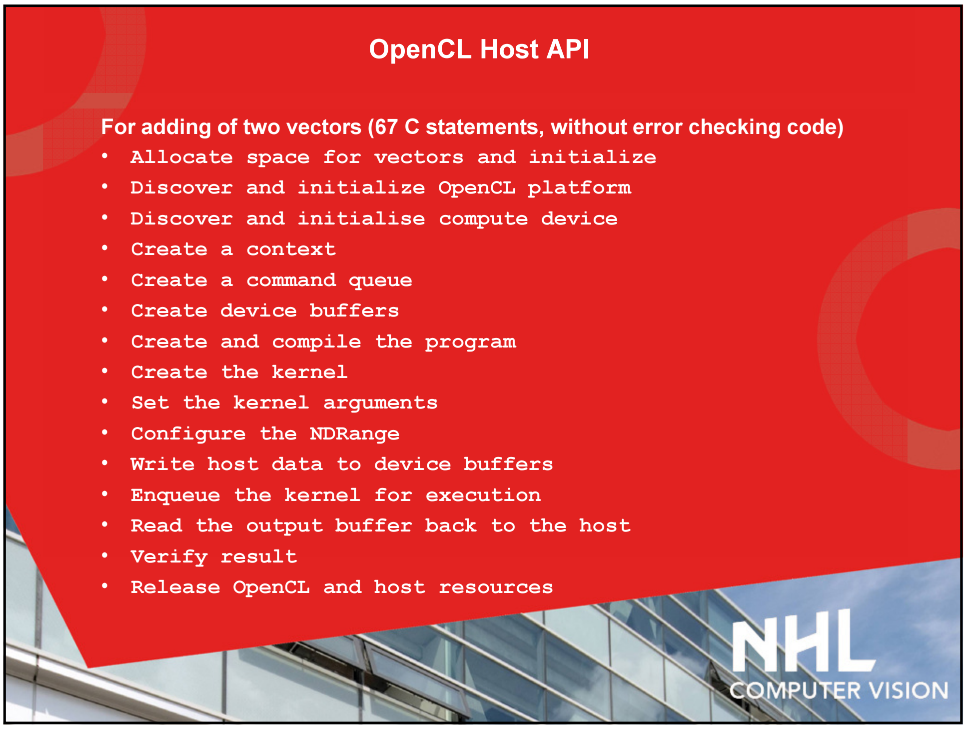
OpenCL execution model



```
__kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
 int id = get_global_id(0);
 c[id] = a[id] * b[id];
}
```

After OpenCL A Standard platform for programming Heterogeneous parallel computers, Mattson et al, 2009

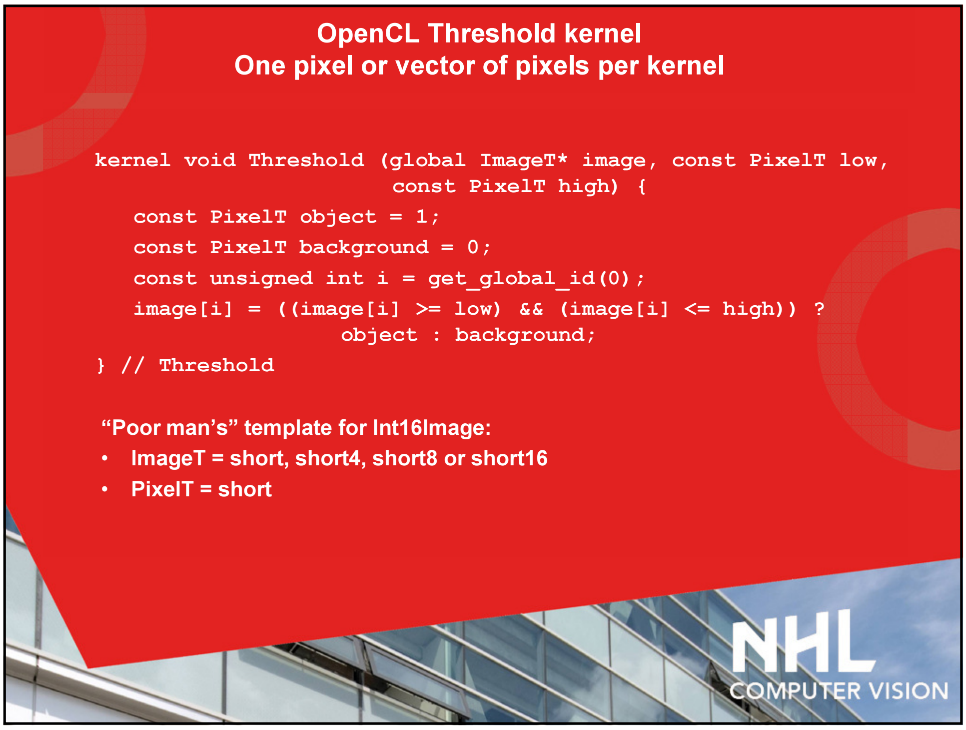
NHL COMPUTER VISION



OpenCL Host API

For adding of two vectors (67 C statements, without error checking code)

- `Allocate space for vectors and initialize`
- `Discover and initialize OpenCL platform`
- `Discover and initialise compute device`
- `Create a context`
- `Create a command queue`
- `Create device buffers`
- `Create and compile the program`
- `Create the kernel`
- `Set the kernel arguments`
- `Configure the NDRange`
- `Write host data to device buffers`
- `Enqueue the kernel for execution`
- `Read the output buffer back to the host`
- `Verify result`
- `Release OpenCL and host resources`



OpenCL Threshold kernel
One pixel or vector of pixels per kernel

```
kernel void Threshold (global ImageT* image, const PixelT low,
                       const PixelT high) {
   const PixelT object = 1;
   const PixelT background = 0;
   const unsigned int i = get_global_id(0);
   image[i] = ((image[i] >= low) && (image[i] <= high)) ?
                   object : background;
} // Threshold
```

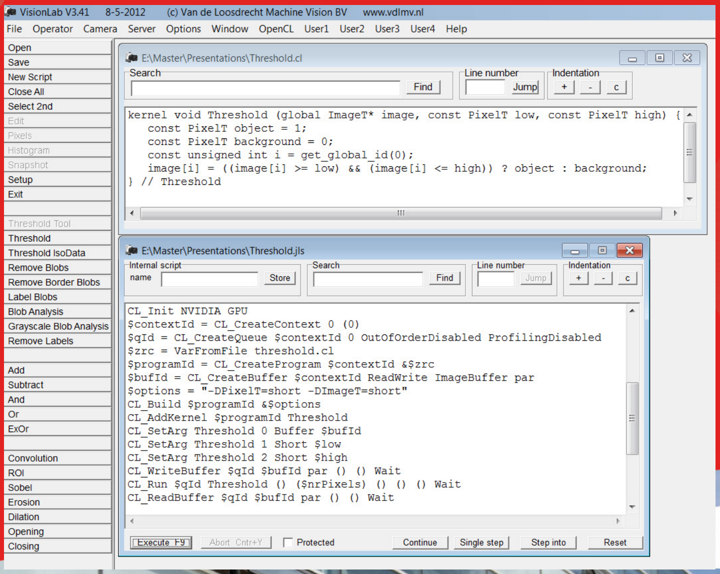
“Poor man’s” template for Int16Image:

- ImageT = short, short4, short8 or short16
- PixelT = short

OpenCL Threshold host side VisionLab script

```
CL_Init NVIDIA GPU
$contextId = CL_CreateContext 0 (0)
$qId = CL_CreateQueue $contextId 0 OutOfOrderDisabled ProfilingDisabled
$zrc = VarFromFile threshold.cl
$programId = CL_CreateProgram $contextId &$zrc
$bufId = CL_CreateBuffer $contextId ReadWrite ImageBuffer par
$options = "-DPixelT=short -DImageT=short"
CL_Build $programId &$options
CL_AddKernel $programId Threshold
CL_SetArg Threshold 0 Buffer $bufId
CL_SetArg Threshold 1 Short $low
CL_SetArg Threshold 2 Short $high
CL_WriteBuffer $qId $bufId par () () Wait
CL_Run $qId Threshold () ($nrPixels) () () () Wait
CL_ReadBuffer $qId $bufId par () () Wait
```

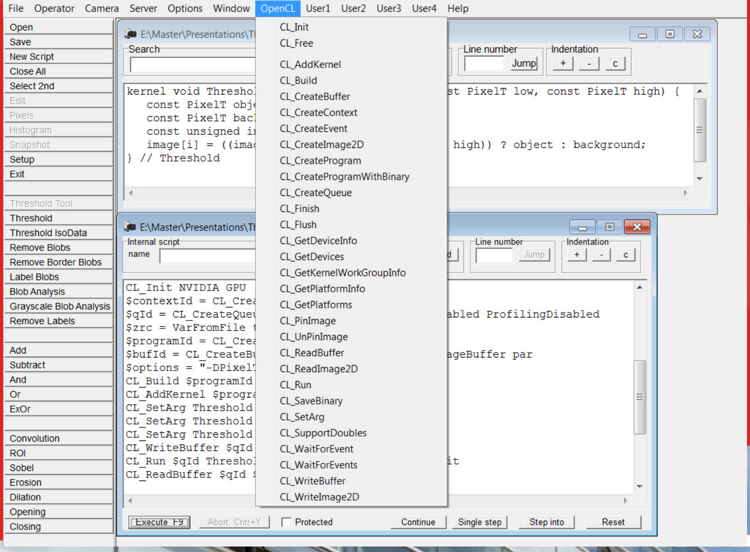
OpenCL development in VisionLab



```
kernel void Threshold (global ImageT* image, const PixelT low, const PixelT high) {
   const PixelT object = 1;
   const PixelT background = 0;
   const unsigned int i = get_global_id(0);
   image[i] = ((image[i] >= low) && (image[i] <= high)) ? object : background;
} // Threshold
```

```
CL_Init NVIDIA GPU
$contextId = CL_CreateContext 0 (0)
$qId = CL_CreateQueue $contextId 0 OutOfOrderDisabled ProfilingDisabled
$zrc = VarFromFile threshold.cl
$programId = CL_CreateProgram $contextId &$zrc
$bufId = CL_CreateBuffer $contextId ReadWrite ImageBuffer par
$options = "-DPixelT=short -DImageT=short"
CL_Build $programId &$options
CL_AddKernel $programId Threshold
CL_SetArg Threshold 0 Buffer $bufId
CL_SetArg Threshold 1 Short $low
CL_SetArg Threshold 2 Short $high
CL_WriteBuffer $qId $bufId par () () Wait
CL_Run $qId Threshold () ($nrPixels) () () () Wait
CL_ReadBuffer $qId $bufId par () () Wait
```

OpenCL development in VisionLab

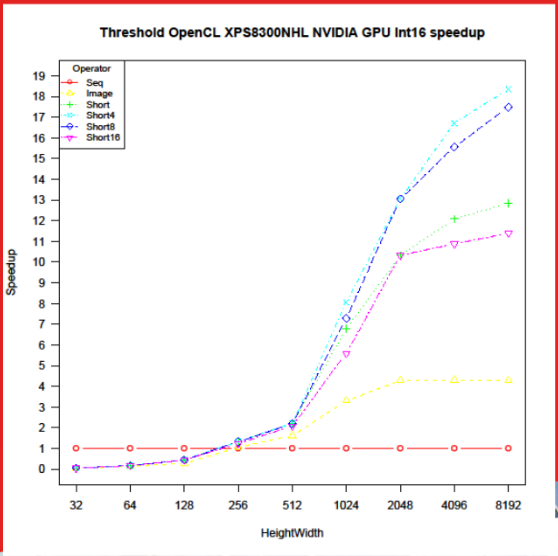


```
kernel void Threshol                       st PixelT low, const PixelT high) {
   const PixelT obje
   const PixelT bacl
   const unsigned ir
   image[i] = ((imac                        high)) ? object : background;
} // Threshold
```

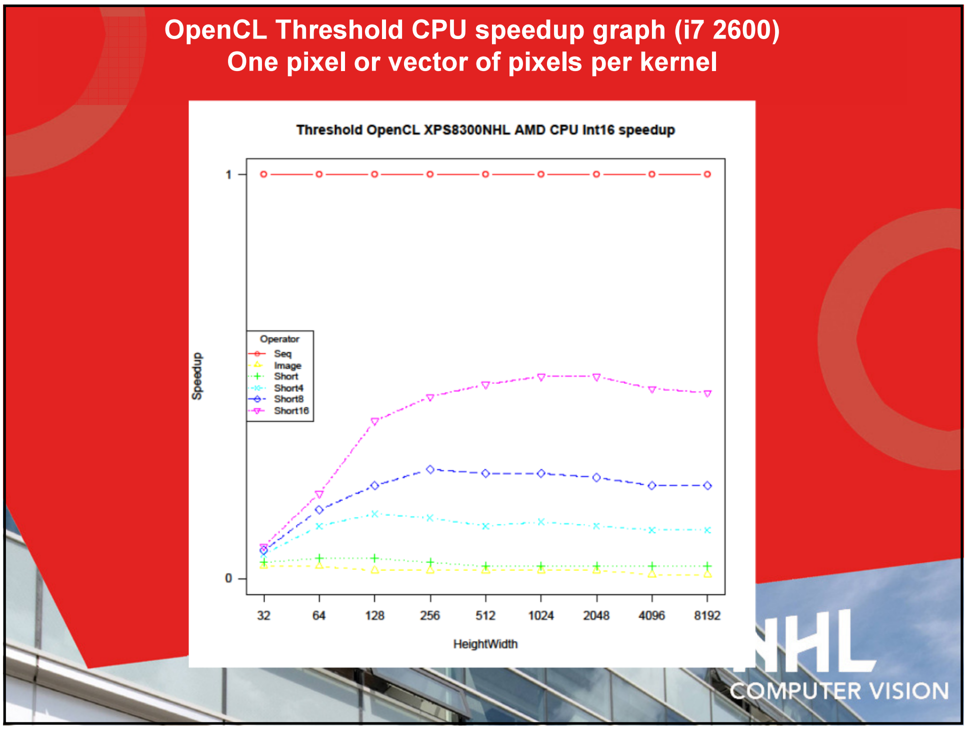
```
CL_Init NVIDIA GPU
$contextId = CL_Crea
$qId = CL_CreateQueu                       abled ProfilingDisabled
$zrc = VarFromFile t
$programId = CL_Crea
$bufId = CL_CreateBu                       ageBuffer par
$options = "-DPixelT
CL_Build $programId
CL_AddKernel $progra
CL_SetArg Threshold
CL_SetArg Threshold
CL_SetArg Threshold
CL_WriteBuffer $qId
CL_Run $qId Threshol                       it
CL_ReadBuffer $qId $
```

COMPUTER VISION

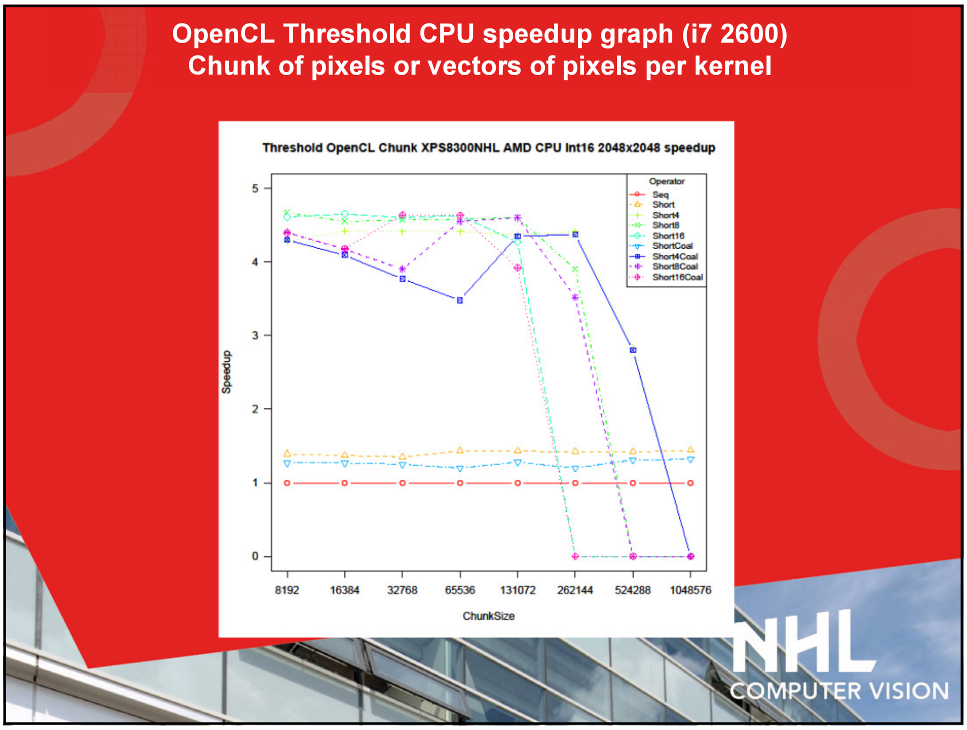
OpenCL Threshold GPU speedup graph (GTX 560 Ti)
One pixel or vector of pixels per kernel



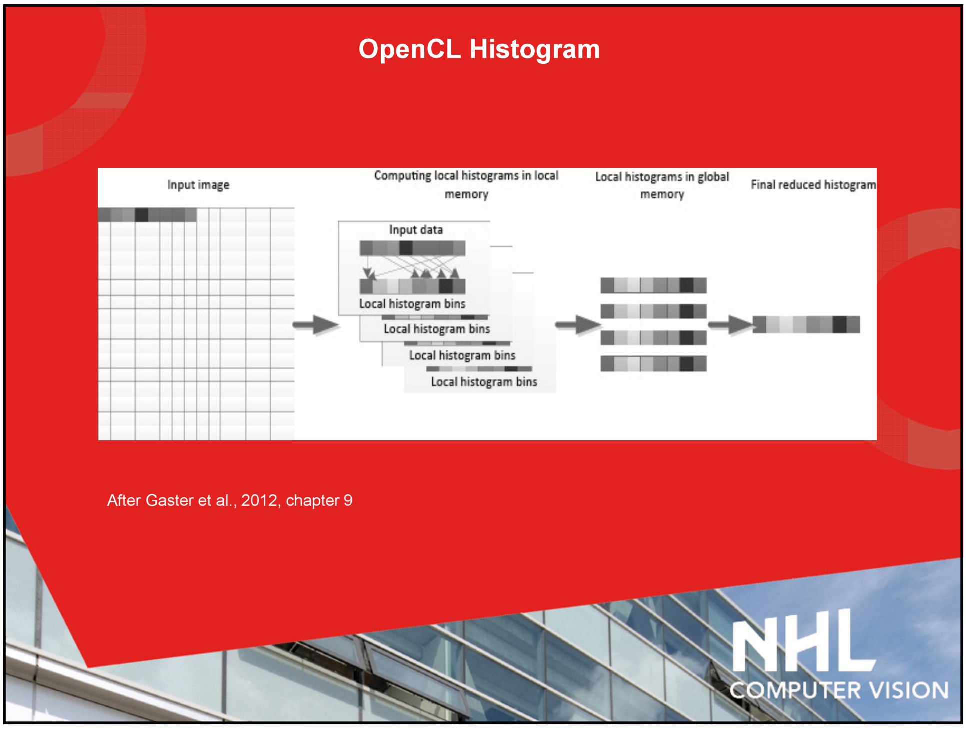
NHL
COMPUTER VISION



OpenCL Threshold CPU speedup graph (i7 2600)
One pixel or vector of pixels per kernel

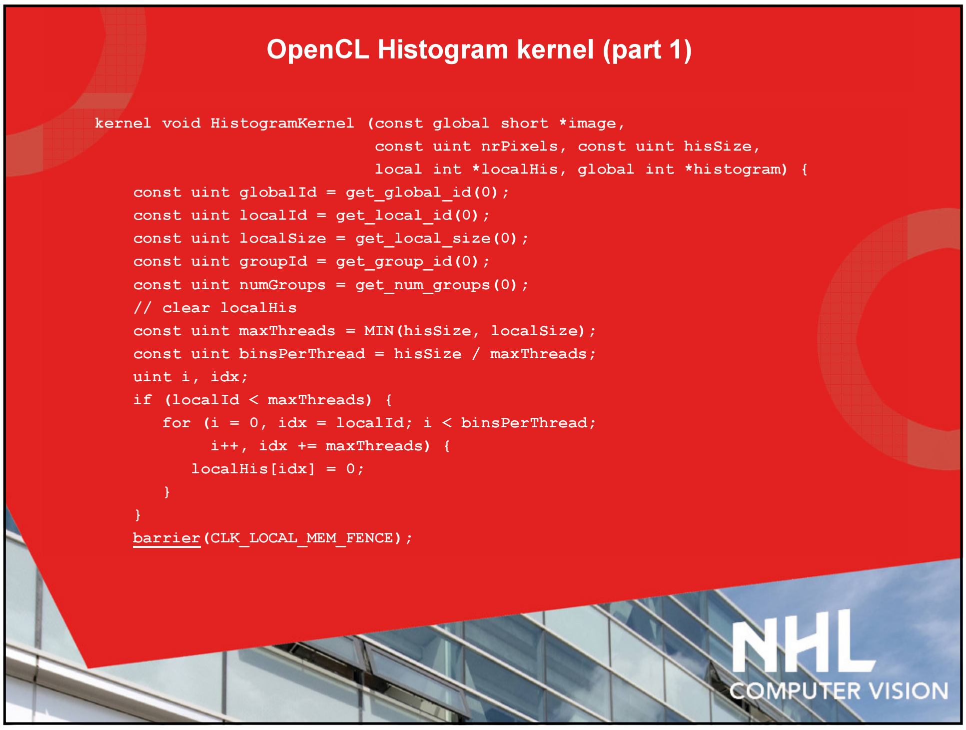


OpenCL Threshold CPU speedup graph (i7 2600)
Chunk of pixels or vectors of pixels per kernel



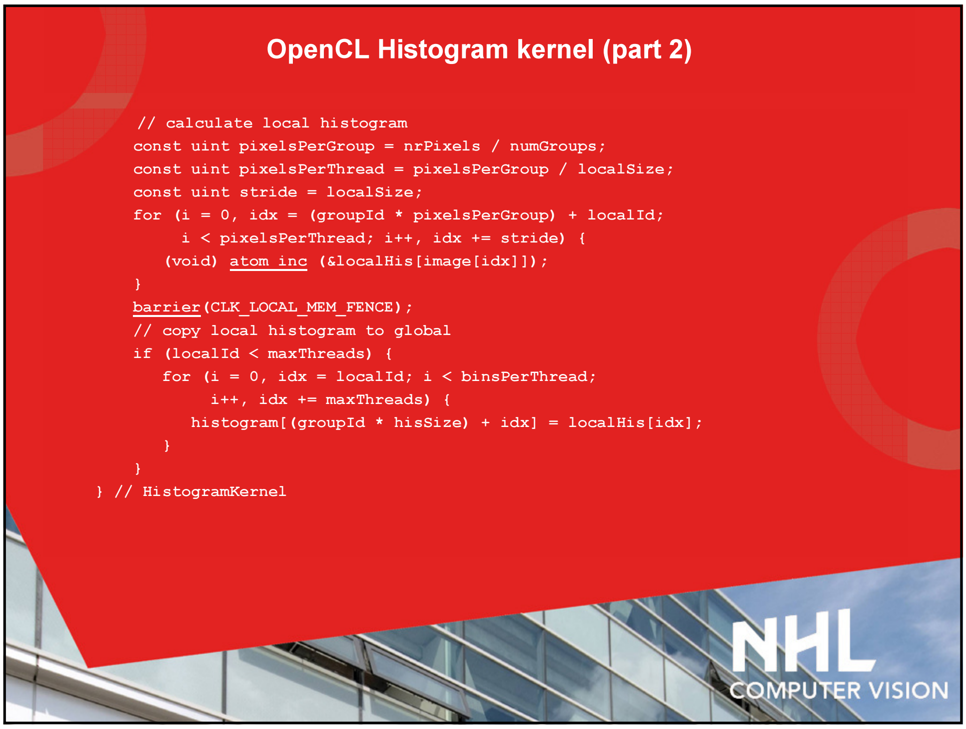
OpenCL Histogram

After Gaster et al., 2012, chapter 9



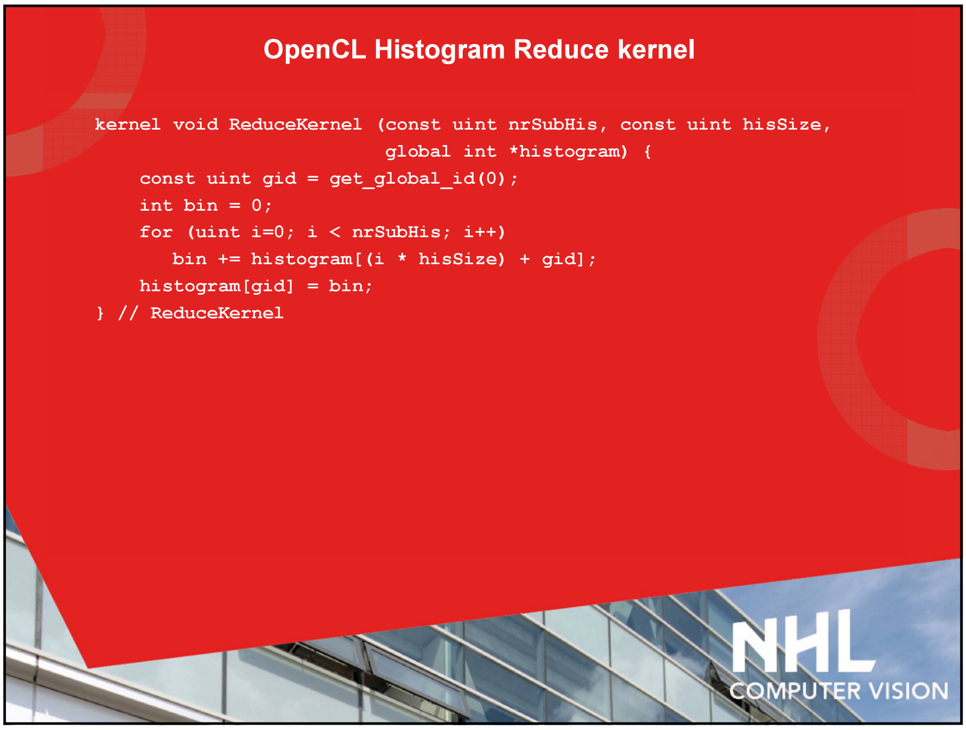
OpenCL Histogram kernel (part 1)

```
kernel void HistogramKernel (const global short *image,
                             const uint nrPixels, const uint hisSize,
                             local int *localHis, global int *histogram) {
    const uint globalId = get_global_id(0);
    const uint localId = get_local_id(0);
    const uint localSize = get_local_size(0);
    const uint groupId = get_group_id(0);
    const uint numGroups = get_num_groups(0);
    // clear localHis
    const uint maxThreads = MIN(hisSize, localSize);
    const uint binsPerThread = hisSize / maxThreads;
    uint i, idx;
    if (localId < maxThreads) {
       for (i = 0, idx = localId; i < binsPerThread;
            i++, idx += maxThreads) {
          localHis[idx] = 0;
       }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
```



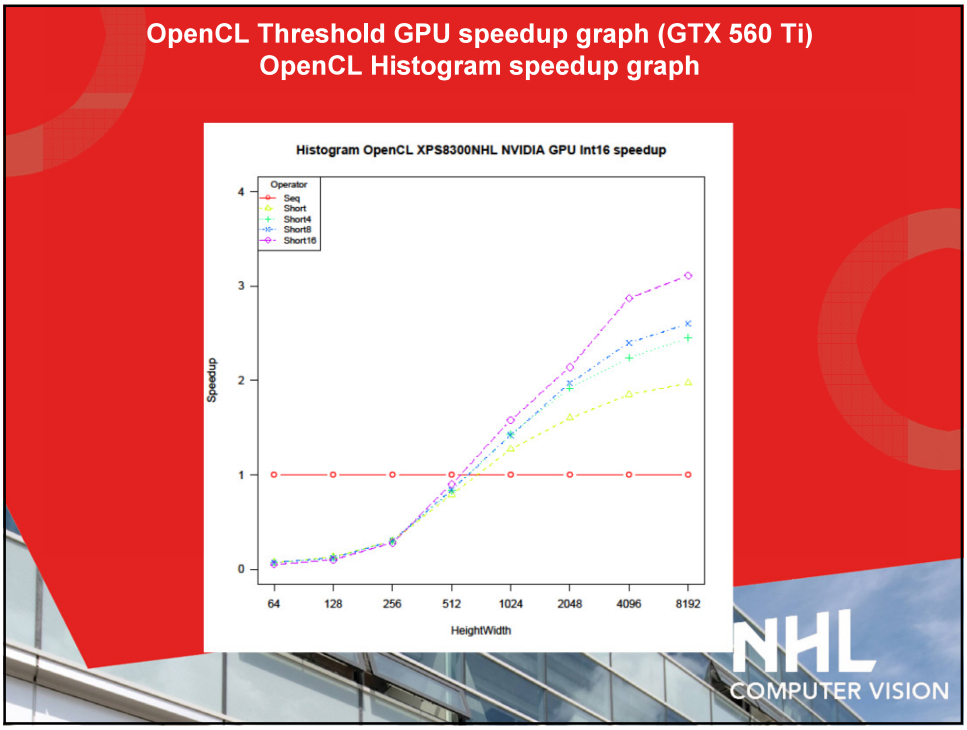
OpenCL Histogram kernel (part 2)

```
    // calculate local histogram
    const uint pixelsPerGroup = nrPixels / numGroups;
    const uint pixelsPerThread = pixelsPerGroup / localSize;
    const uint stride = localSize;
    for (i = 0, idx = (groupId * pixelsPerGroup) + localId;
         i < pixelsPerThread; i++, idx += stride) {
       (void) atom_inc (&localHis[image[idx]]);
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    // copy local histogram to global
    if (localId < maxThreads) {
       for (i = 0, idx = localId; i < binsPerThread;
            i++, idx += maxThreads) {
          histogram[(groupId * hisSize) + idx] = localHis[idx];
       }
    }
} // HistogramKernel
```

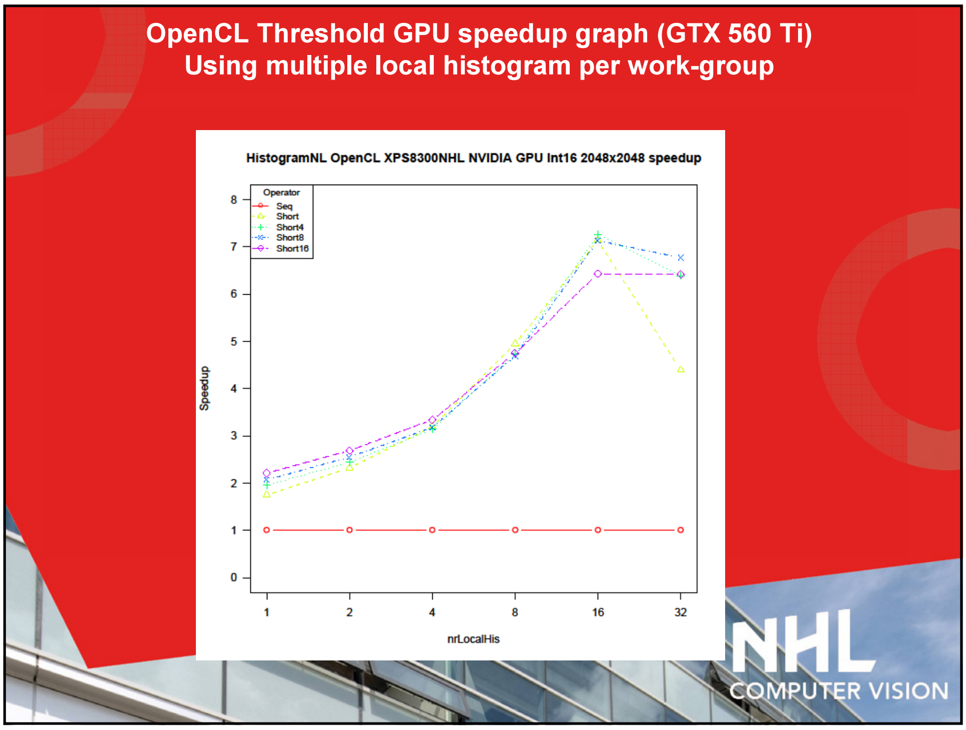


OpenCL Histogram Reduce kernel

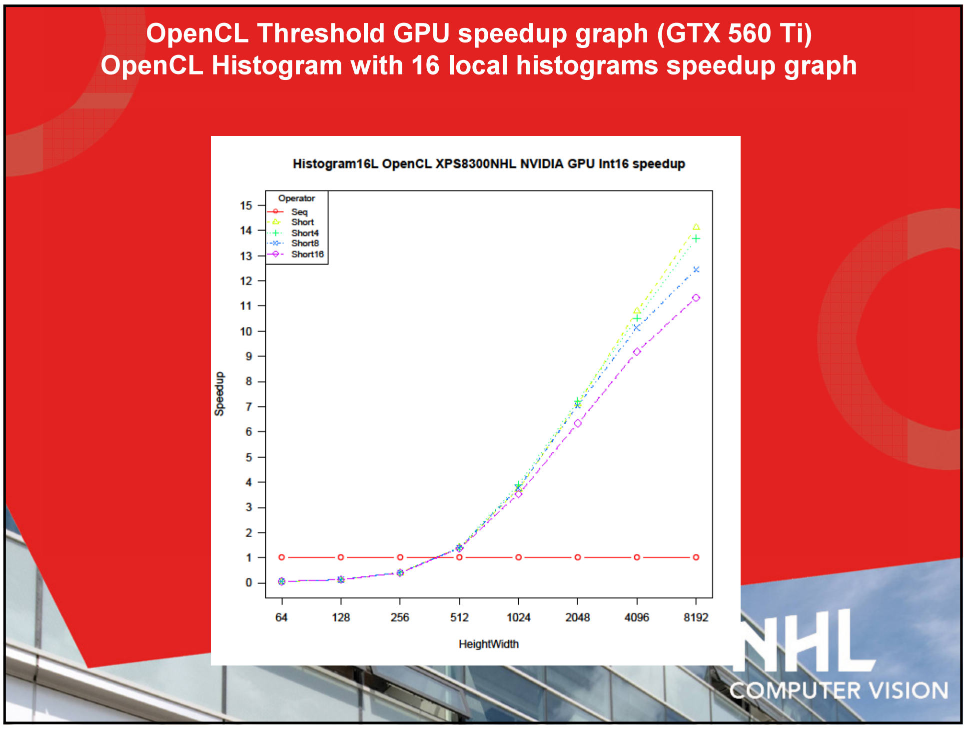
```
kernel void ReduceKernel (const uint nrSubHis, const uint hisSize,
                          global int *histogram) {
    const uint gid = get_global_id(0);
    int bin = 0;
    for (uint i=0; i < nrSubHis; i++)
       bin += histogram[(i * hisSize) + gid];
    histogram[gid] = bin;
} // ReduceKernel
```



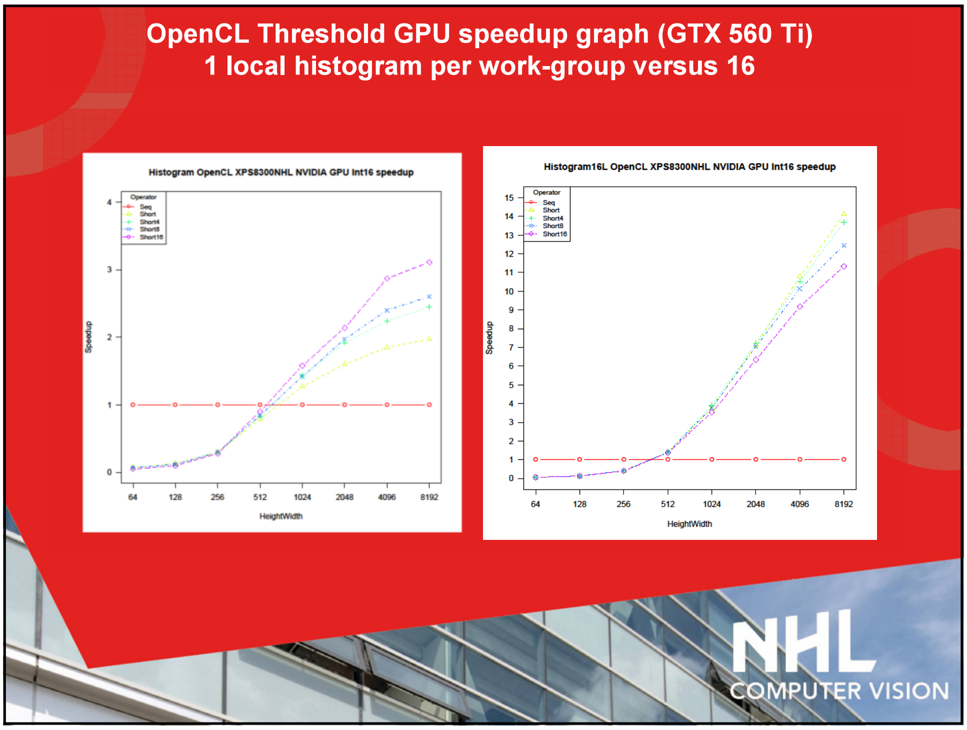
OpenCL Threshold GPU speedup graph (GTX 560 Ti)
OpenCL Histogram speedup graph



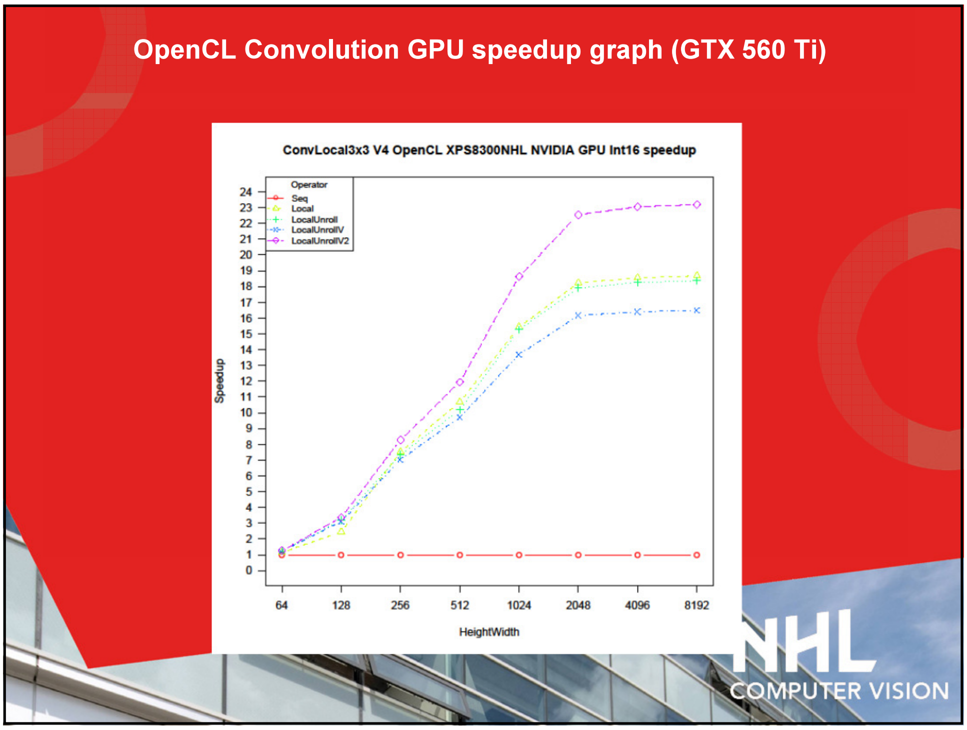
OpenCL Threshold GPU speedup graph (GTX 560 Ti)
Using multiple local histogram per work-group



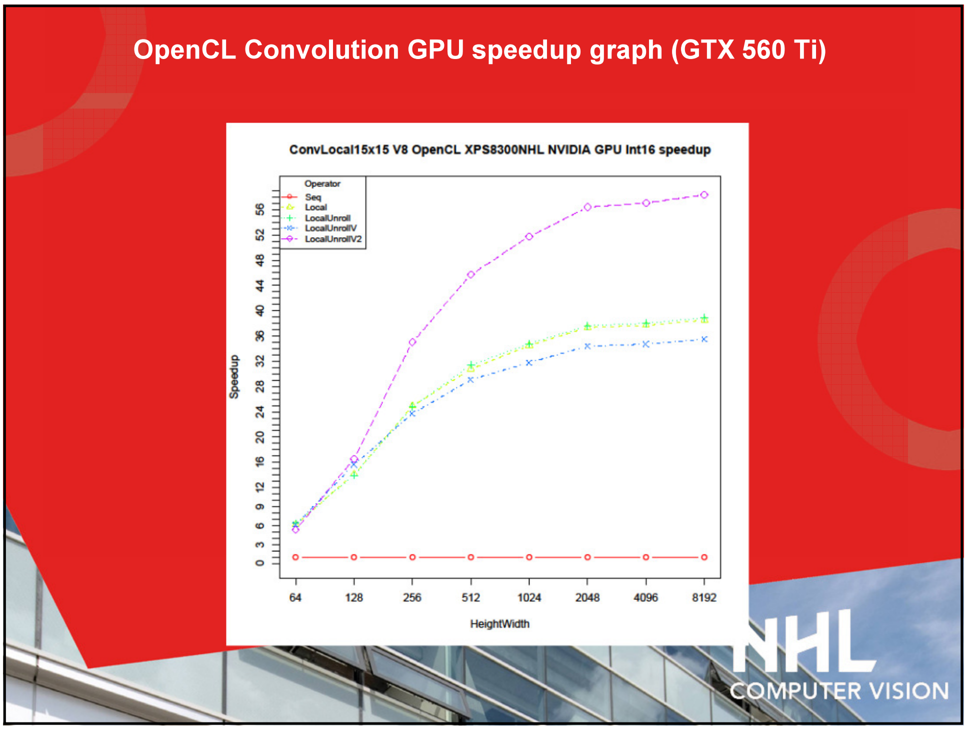
OpenCL Threshold GPU speedup graph (GTX 560 Ti)
OpenCL Histogram with 16 local histograms speedup graph



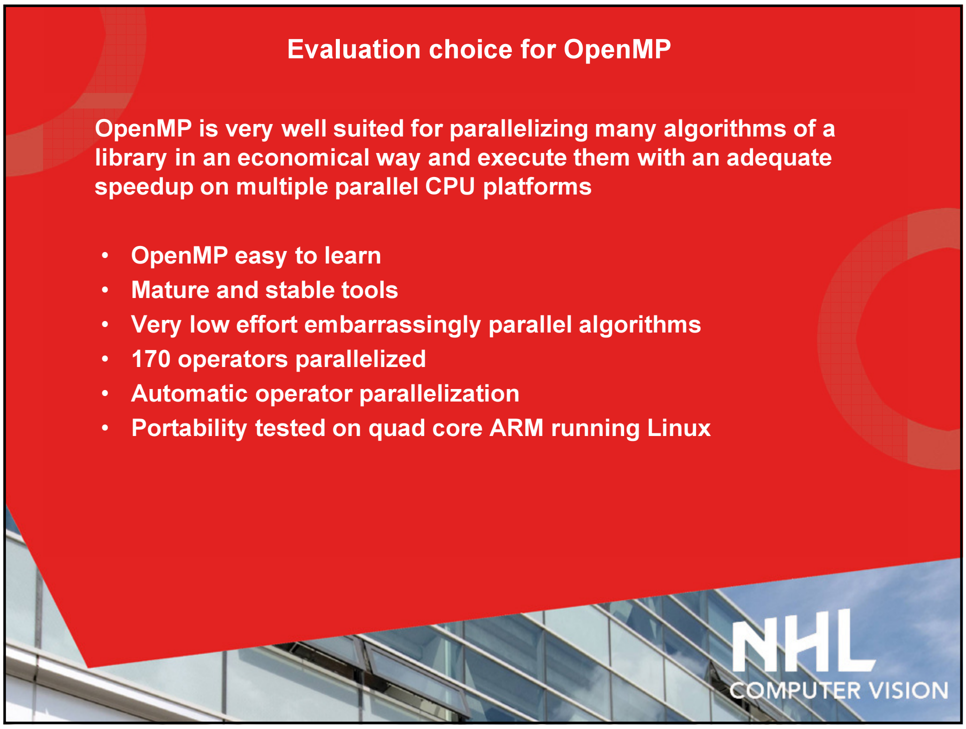
OpenCL Threshold GPU speedup graph (GTX 560 Ti)
1 local histogram per work-group versus 16



OpenCL Convolution GPU speedup graph (GTX 560 Ti)



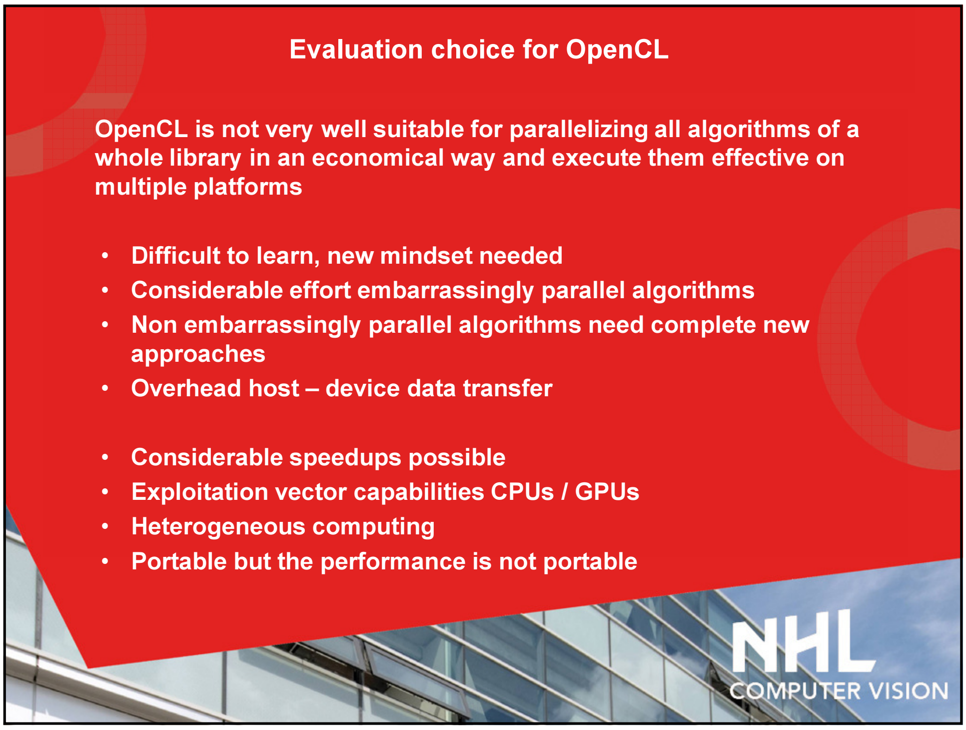
OpenCL Convolution GPU speedup graph (GTX 560 Ti)



Evaluation choice for OpenMP

OpenMP is very well suited for parallelizing many algorithms of a library in an economical way and execute them with an adequate speedup on multiple parallel CPU platforms

- OpenMP easy to learn
- Mature and stable tools
- Very low effort embarrassingly parallel algorithms
- 170 operators parallelized
- Automatic operator parallelization
- Portability tested on quad core ARM running Linux

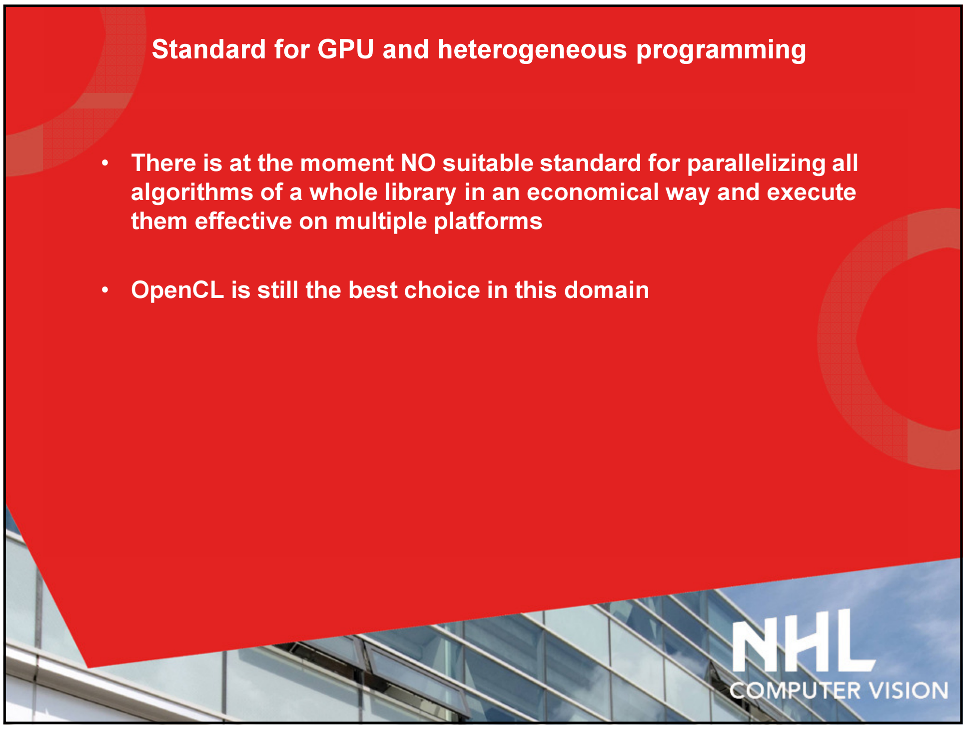


Evaluation choice for OpenCL

OpenCL is not very well suitable for parallelizing all algorithms of a whole library in an economical way and execute them effective on multiple platforms

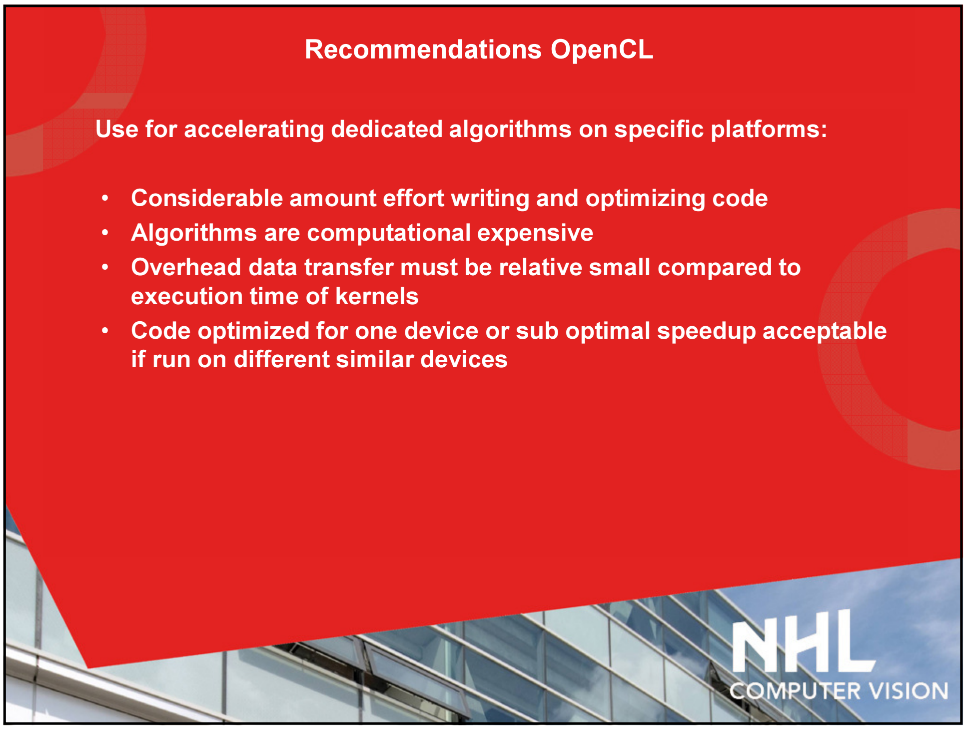
- Difficult to learn, new mindset needed
- Considerable effort embarrassingly parallel algorithms
- Non embarrassingly parallel algorithms need complete new approaches
- Overhead host – device data transfer

- Considerable speedups possible
- Exploitation vector capabilities CPUs / GPUs
- Heterogeneous computing
- Portable but the performance is not portable



Standard for GPU and heterogeneous programming

- There is at the moment NO suitable standard for parallelizing all algorithms of a whole library in an economical way and execute them effective on multiple platforms
- OpenCL is still the best choice in this domain



Recommendations OpenCL

Use for accelerating dedicated algorithms on specific platforms:

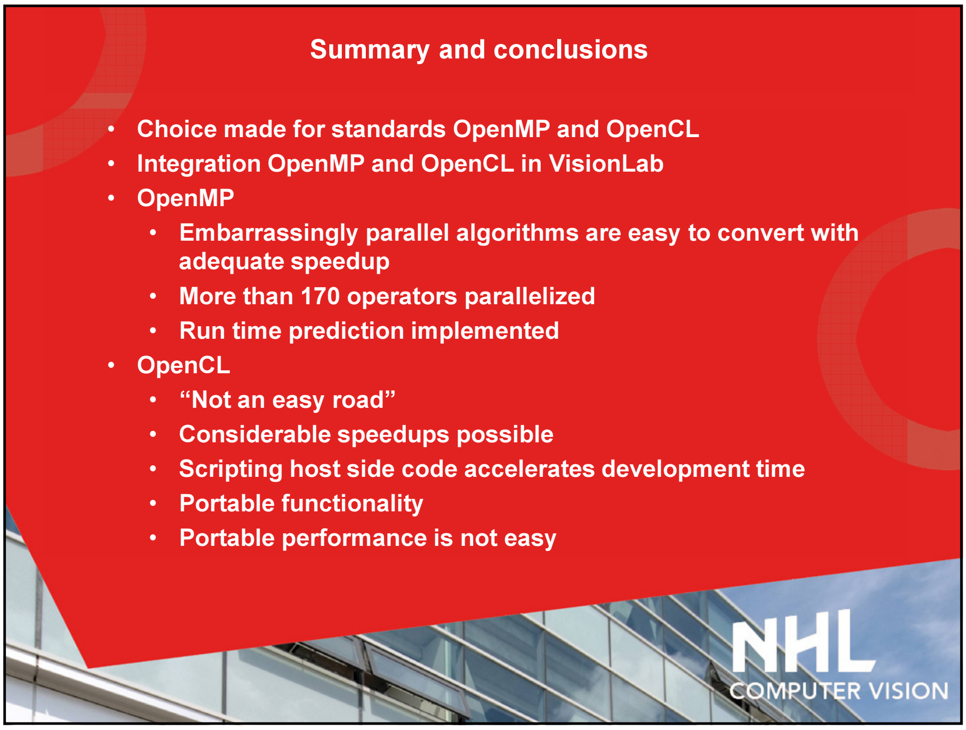
- Considerable amount effort writing and optimizing code
- Algorithms are computational expensive
- Overhead data transfer must be relative small compared to execution time of kernels
- Code optimized for one device or sub optimal speedup acceptable if run on different similar devices

Future work

New developments in standards
- C++ AMP
- OpenMP 4.0

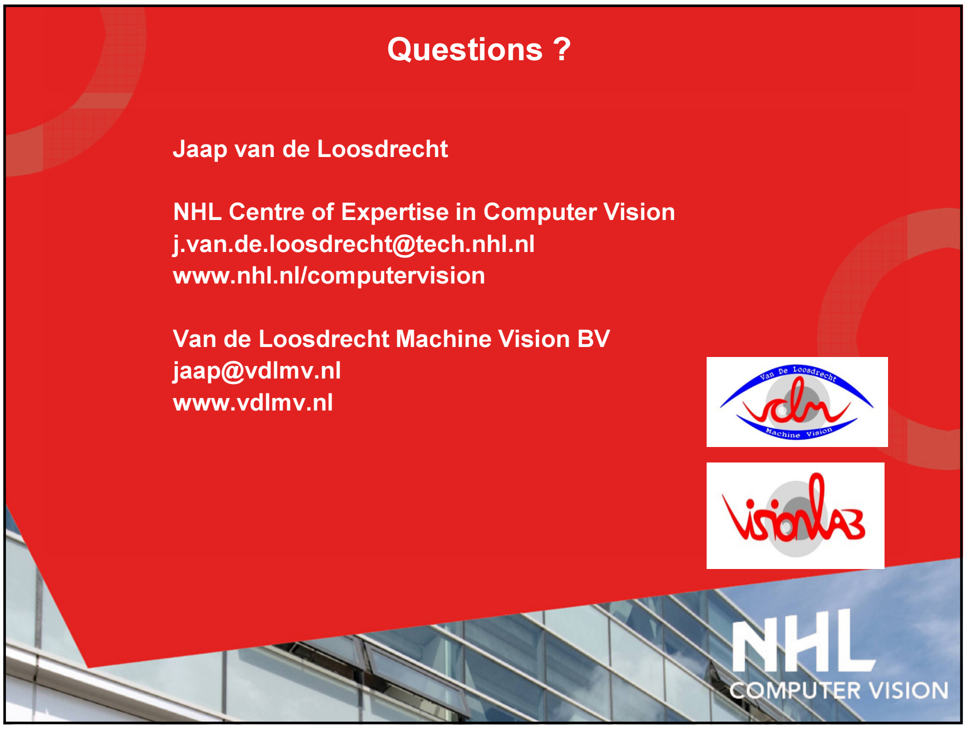
Near future
- Parallelize more vision operators

More distant future
- Intelligent buffer management
- Automatic tuning of parameters
- Heterogeneous computing



Summary and conclusions

- Choice made for standards OpenMP and OpenCL
- Integration OpenMP and OpenCL in VisionLab
- OpenMP
  - Embarrassingly parallel algorithms are easy to convert with adequate speedup
  - More than 170 operators parallelized
  - Run time prediction implemented
- OpenCL
  - “Not an easy road”
  - Considerable speedups possible
  - Scripting host side code accelerates development time
  - Portable functionality
  - Portable performance is not easy



Questions ?

Jaap van de Loosdrecht

NHL Centre of Expertise in Computer Vision
j.van.de.loosdrecht@tech.nhl.nl
www.nhl.nl/computervision

Van de Loosdrecht Machine Vision BV
jaap@vdlmv.nl
www.vdlmv.nl