Accelerating sequential computer vision algorithms using OpenMP and OpenCL on commodity parallel hardware

10 April 2018

j.van.de.loosdrecht@nhl.nl, jaap@vdlmv.nl

Overview

- **Why go parallel ? ***
- **Introduction CPU and GPU architecture**
- **Speedup factor; Amdahl's and Gustafson's law ***
- **Survey of 22 standards for parallel programming**
- **Classification of low level image operators ***
- **OpenMP**
- **OpenCL**
- **Implementation examples**
- **Evaluation choice for OpenMP and OpenCL**
- **Future work ***
- **Summary and conclusions**

Overview

- **Introduction CPU and GPU architecture**
- **Survey of 22 standards for parallel programming**
- **OpenMP**
- **OpenCL**
- **Implementation examples**
- **Evaluation choice for OpenMP and OpenCL**
- **Summary and preliminary conclusions**

Why go parallel?

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

Parallelizing VisionLab

Aims and objectives:

- **Compare existing programming languages and environments for parallel computing**
- **Choose one standard for Multi-core CPU programming and for GPU programming**
- **Re-implement a number of standard and well-known algorithms**
- **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 Computer Vision 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**
    See lecture "Multi-core CPU processing in VisionLab"

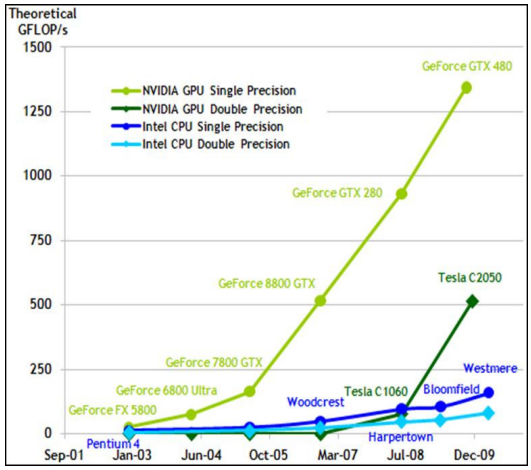
Introduction CPU and GPU architecture

Observations:

- **The last years CPUs do not much become faster than about 4 GHz**
- **Multi-core CPU PCs are now widely available at low costs**
- **Graphics cards (GPUs) have much more computing power than CPUs and are available at low costs**

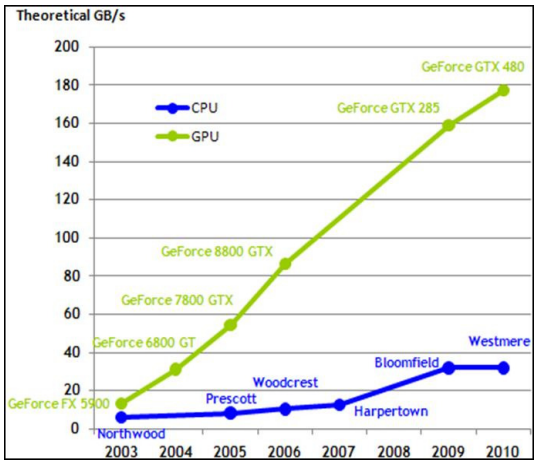
"*The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*" Sutter (2005) predicted that the only way to get more processing power in the future, is to go for parallel programming, and that it is not going to be an easy way

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

27-8-2018 OpenMP and OpenCL 9

Floating point operations per second comparison between CPU and GPU

After The Heterogeneous System architecture its (not) all about the GPU, Blinzer, P., 2014

27-8-2018 OpenMP and OpenCL 10

GPU vs CPU

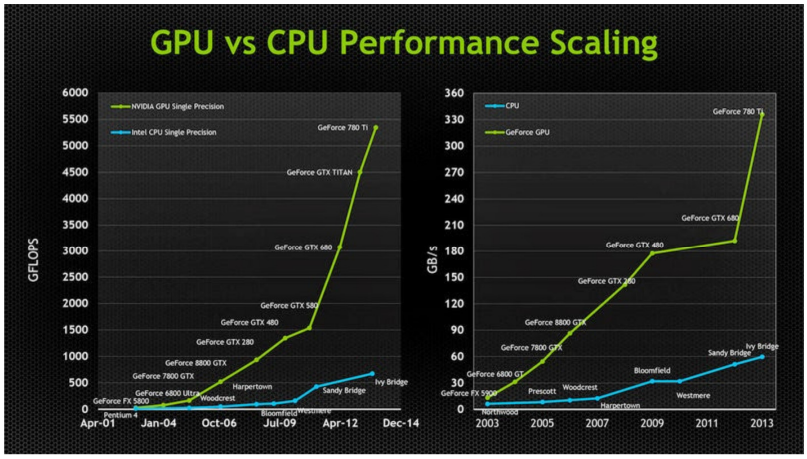


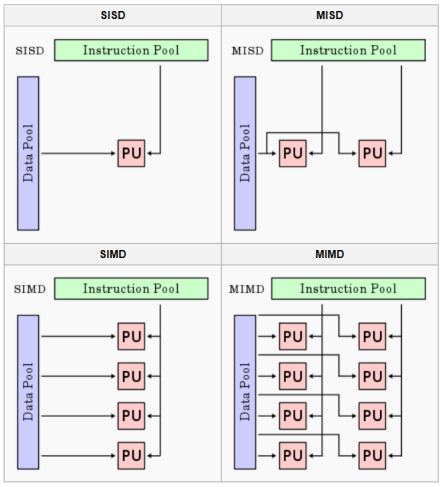
Image Source: nvidia.com

Flynn's taxonomy

- **Single Instruction, Single Data stream (SISD)**
- **Single Instruction, Multiple Data stream (SIMD)**
- **Multiple Instruction, Single Data stream (MISD)**
- **Multiple Instruction, Multiple Data stream (MIMD)**

Wavefront array architectures (GPUs) are a specialization of SIMD

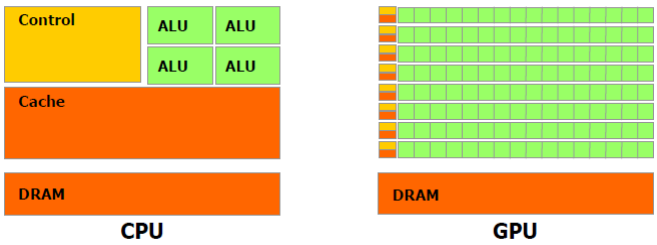
Flynn's taxonomy



After en.wikipedia.org/wiki/Flynn's_taxonomy

27-8-2018 OpenMP and OpenCL 13

Usage of transistors on chip



After OpenCL programming guide for CUDA architecture, NVIDIA, 2010

27-8-2018 OpenMP and OpenCL 14

CPU architecture

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

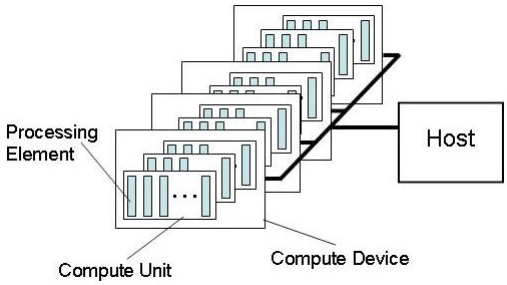
- **Multi-core MIMD architecture with**
  - **Branch prediction**
  - **Out-of-order execution**
  - **Super-scalar**
  - **Each core SIMD vector processor**
  - **Complex hierarchy of cache memory with cache coherence**
- **Restricted by thermal envelope**

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 ***

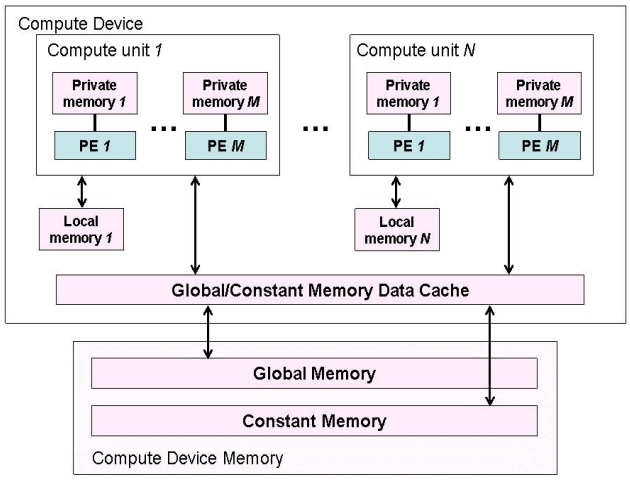
Host and device



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

27-8-2018 OpenMP and OpenCL 17

Device architecture (OpenCL)



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

27-8-2018 OpenMP and OpenCL 18

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

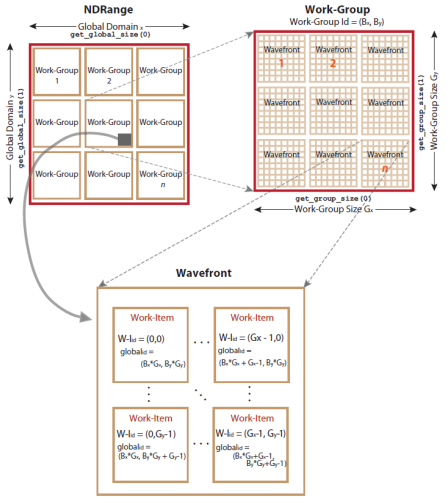
AMD R9 290

- **40 compute units with each 4 processor elements**
- **Each processor element is a 16-way SIMD like vector processor**
- **One compute unit = 4 x 16 = 64 sub-processors**
- **40 compute units = 2560 sub-processors**
- **Running at 950 Mhz, a peak performance of 4.9 TFlops.**
- **4 GByte global memory, peak bandwidth of 320 GBytes/s**
- **64 KByte local memory for each compute unit**
- **64 KByte private memory for each processor element**

Warps or wavefronts

- Work-items are organised in workgroups
- Work-items in a workgroup are organized in warps
- A work-item has
  - Global id
  - Workgroup id
  - Local id

NDRange, Work-Groups and Wavefronts

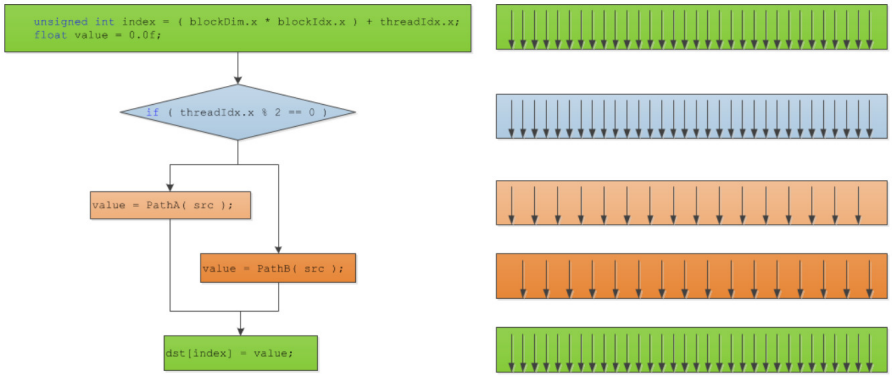


After AMD Accelerated Parallel Processing
OpenCL Programming GuideAMD, 2011

Warps or wavefronts

- All work-items in a warp execute the same instruction in parallel on all cores of a compute unit in SIMT fashion
- Single Instruction Multiple Thread (SIMT): vector components are considered as individual threads that can branch independently
- Typical size for a warp is 16, 32 or 64 work-items
- Branch divergence

Branch divergence



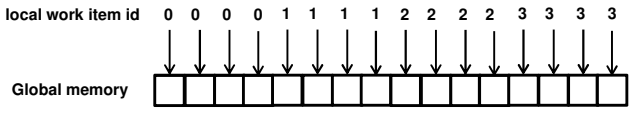
After Optimizing CUDA Applications, van Oosten, J., 2011

Warps or wavefronts

- **All work-items in a warp execute the same instruction in parallel on all cores of a compute unit in SIMT fashion**
- **Single Instruction Multiple Thread (SIMT): vector components are considered as individual threads that can branch independently**
- **Typical size for a warp is 16, 32 or 64 work-items**
- **Branch divergence**
- **Occupancy rate**
  Hide long global memory latency
- **Zero-overhead warp scheduling**

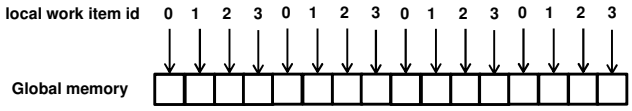
Non-coalesced memory access of global memory

Example for warp with four work-items and chunk size = 4



Coalesced memory access of global memory

Example for warp with four work-items and chunk size = 4



27-8-2018 OpenMP and OpenCL 27

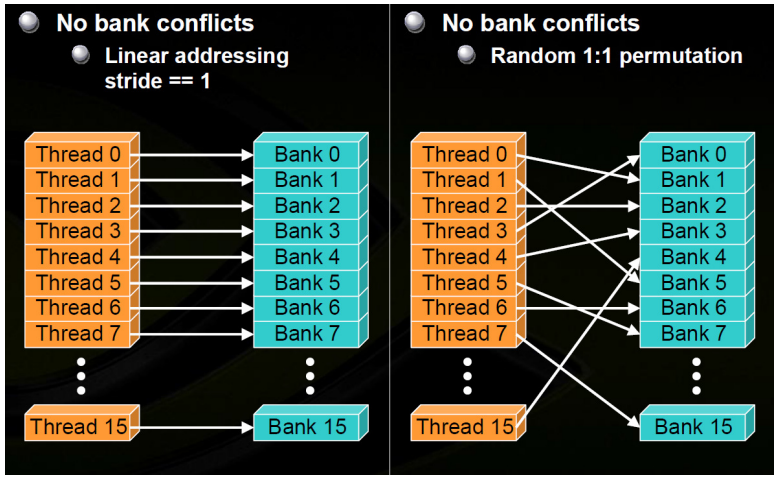
Bank conflicts in accessing local memory

In order to access fast local memory

- **Divided in banks**
- **Dedicated channels**
- **Successive N words are assigned to successive banks**

27-8-2018 OpenMP and OpenCL 28

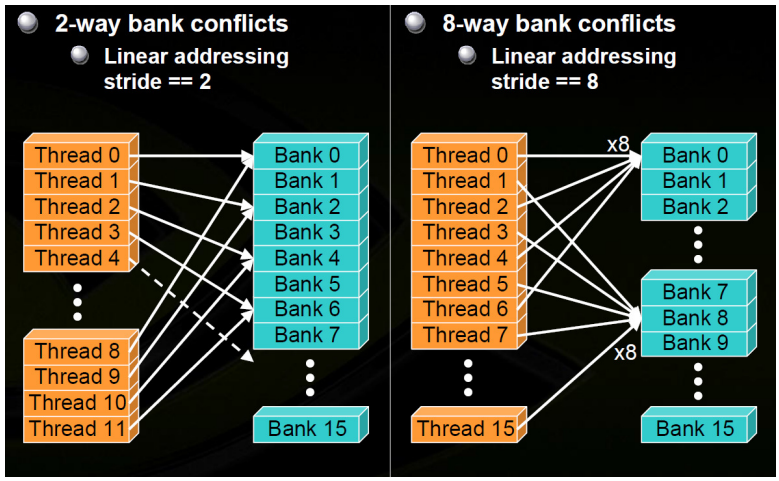
Bank conflicts in accessing local memory



After CUDA Tutorial, NVIDIA, 2008

27-8-2018 OpenMP and OpenCL 29

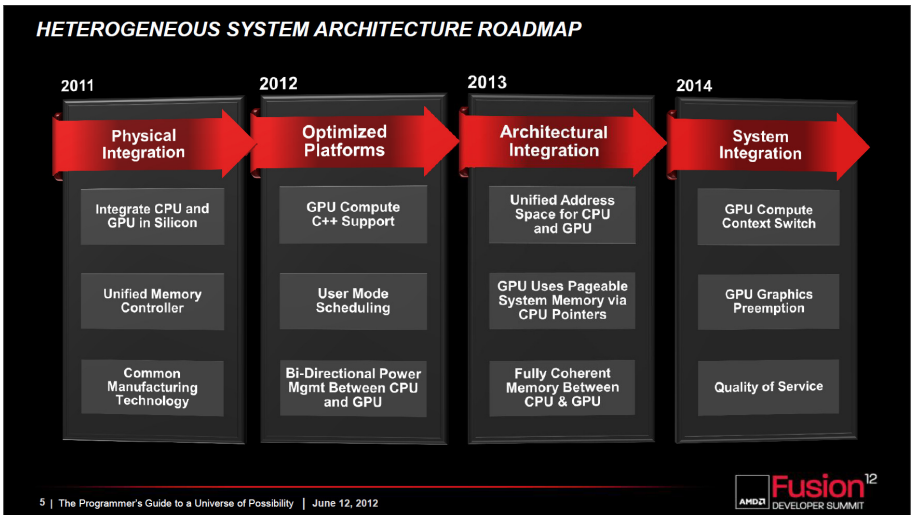
Bank conflicts in accessing local memory



After CUDA Tutorial, NVIDIA, 2008

27-8-2018 OpenMP and OpenCL 30

HSA Roadmap



After P. Rogers, AMD 2012

Speedup factor

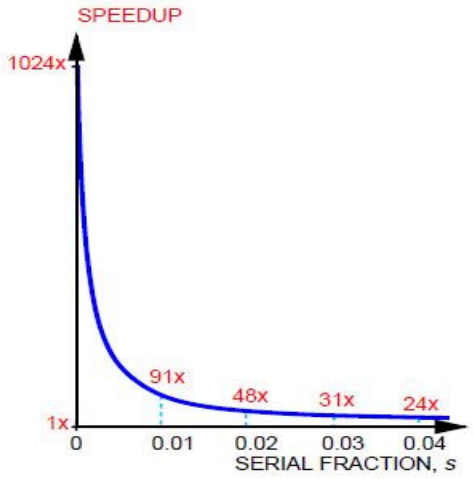
T_1 = execution time with one core
T_N = execution time with N cores
Speedup factor = T_1 / T_N

P = time spent in the fraction of the program that can be parallelized
S = time spent in the serial fraction

$$Speedup_{Amdahl} = \frac{S+P}{S+P/N} = \frac{1}{(1-P)+P/N} = \frac{1}{S+(1-S)/N}$$

Example: if 80% of the code can be parallelized then the speedup <= 5

Amdahl's law: speedup for 1024 cores



After: Gustafson, Montry and Benner, 1988

27-8-2018 OpenMP and OpenCL 33

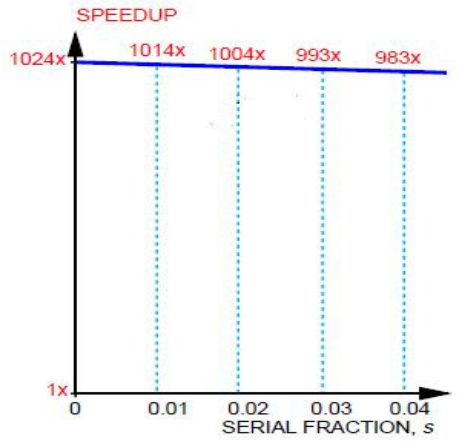
Gustafson's law

- **More processors and memory: many problems are scaled with N**
- **Problem is scaled: in many cases S decreases**
- **Amdahl's law: how fast a given serial program would run on a parallel system**
- **Gustafson's law: how long a given parallel program would run on a sequential processor**

$$Speedup_{Gustafson} = \frac{S + P * N}{S + P} = N - (N - 1) * S$$

27-8-2018 OpenMP and OpenCL 34

Gustafson's law: speedup for 1024 cores



After: Gustafson, Montry and Benner, 1988

27-8-2018 OpenMP and OpenCL 35

Heterogeneous architecture

Heterogeneous = combination of CPU and GPU

Example:

- **10% of code can not be parallelized**
- **1 core needs 10 times as many resources to be 2 x faster**

- **$Speedup_{Amdahl}$ with 100 simple cores**
  1 / (0.1 + 0.9/100) = 9.2
- **$Speedup_{Amdahl}$ with 90 simple cores and one 2 x faster core**
  1 / (0.1/2 + 0.9/90) = 16.7

After Asanovic et al., 2006

27-8-2018 OpenMP and OpenCL 36

Survey (2013) 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 standard for multi-core CPU programming

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	Unknown	Good	Good	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 standard for GPU programming

Requirement ---------- Standard	Industry standard	Maturity	Acceptance by market	Future developments	Expected familiarization time	Hardware vendor independence	Software vendor independence	Portability	Heterogeneous
C++ AMP	No	New	Not ranked	Unknown	Medium	Bad	Bad	Poor	No
CUDA	No	Excellent	Rank 5	Good	High	Reasonable	Reasonable	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**	**Good**	**Good**	**Good**	**Yes**
PGI Accelerator	No	Reasonable	Not ranked	Unknown	Medium	Bad	Bad	Bad	No

27-8-2018 OpenMP and OpenCL 39

Acceptance by market

DEVELOPERS PREFER OPEN STANDARDS!

- June 2011 developer survey shows inevitable success and adoption of OpenCL
- Respondents ranked most popular APIs for Multi-Threaded Development
- OpenCL *ALREADY* #2 in N.A., #3 in EMEA

APIs for Current Multi-Threaded Development

The most popular multi-threaded development API used by developer. survey is OpenMP (Open Multi-Processing), which supports multi-platform sh memory multiprocessing programming in C, C++, and FORTRAN. OpenMP currently used by 31% of respondents. OpenCL (Open Computing Language), framework for writing programs that execute across various processor platform follows at 28%. Another 25% use Intel Threading Building Blocks, a C++ templ library that leverages Intel's multi-core processors.

Which of the following do you program with today?	Count	Percent of Responses	Percent of Cases
OpenMP	91	13.9	31.3
OpenCL	81	12.4	27.8
Intel Threading Building Blocks	72	11.0	24.7
Intel Parallel Building Blocks	65	10.0	22.3
CUDA	59	9.0	20.1
Intel Cilk Plus	56	8.6	19.1
MPI	50	7.7	17.1
Co Array Fortran	34	5.2	11.6
Other	145	22.2	49.5
----	----	----	----
Total Responses	653	100	222.9

North American Development Survey: Vol. I, © 2011 Evans Data Corp.

Note that this multiple response question allowed the developers to select as many responses as had, and thus the total number of cases will not come to 100%. The response column shows the total responses, while the case column shows the percent of actual developers (cases) who

Market data provided by Evans Data Corporation | June 2011

7 | AMD Fusion Developer Summit | June 14, 2011

AMD Fusion 11 DEVELOPER SUMMIT

After R, Bergman, AMD 2011

27-8-2018 OpenMP and OpenCL 40

New developments

- **CUDA less vendor/platform specific, but no industry standard**
- **Visual Studio C++ AMP, great tool but vendor specific**
- **Enhancement OpenCL kernel language, C++ like features like classes and templates**
- **Altera Corporation OpenCL program for FPGAs**
- **OpenACC announced and products available**
- **OpenMP 4.0 with "directives for attached accelerators"**
  Portable OpenMP pragma style programming on multi-core CPUs and GPUs, utilize vector capabilities of CPUs and GPUs

Classification of low level image operators

Classification by Nicolescu and Jonker (2001) and Kiran, Anoop and Kumar (2011)

Class	Example
Point operator	Threshold
Global operator	Histogram
Local neighbour operator	Convolution
Connectivity based operator	Label Blobs

Parallelizing a large generic computer vision library

Idea to parallelize large part of the library:

- **One instance of each low level class is implemented and can be used as skeleton to implement other instances of class**
- **Many high level operators are built on the low level operators**

Other operators are special and need will need a dedicated approach to parallelizing

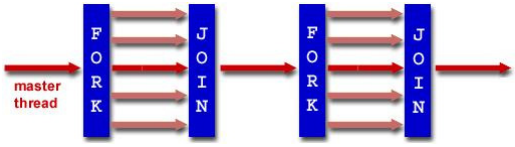
OpenMP

- **Introduction**
- **Components**
- **Scheduling strategy ***
- **Memory model ***
- **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

27-8-2018 OpenMP and OpenCL 45

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 parallel 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**

27-8-2018 Multi Core Processing in VisionLab 46

OpenMP Components

OpenMP consists of three major components:

- **Compiler directives**
- **Runtime functions and variables**
- **Environment variables**

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**

OpenMP runtime functions and variables

OpenMP has runtime functions for performing operations like:

- **Locking**
- **Querying and setting the number of threads to be used in parallel regions**
- **Time measurement**
- **Setting the scheduling strategy**

With environment variables it is possible to modify the default behaviour of OpenMP, like:

- **Setting the maximal number of threads to be used in parallel regions**
- **Setting the stack size for the threads**
- **Setting the scheduling strategy**

OpenMP scheduling strategy

- **Static: iterations are divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.**
- **Dynamic: iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed .**
- **Guided: iterations are assigned to threads in the team in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. The size of the chunk decreases each time.**
- **Auto: decision regarding scheduling is delegated to the compiler and/or runtime system.**

OpenMP memory model

The OpenMP API provides a relaxed-consistency, shared-memory model:

- **Each thread is allowed to have its own temporary view of the memory. The memory model has relaxed-consistency because a thread's temporary view of memory is not required to be consistent with memory at all times.**
- **A flush operation enforces consistency between the temporary view and memory.**

The flush operation:

- **Can be specified using the flush directive**
- **Is implied at:**
  - **A barrier region**
  - **At entry to and exit from parallel and critical region**

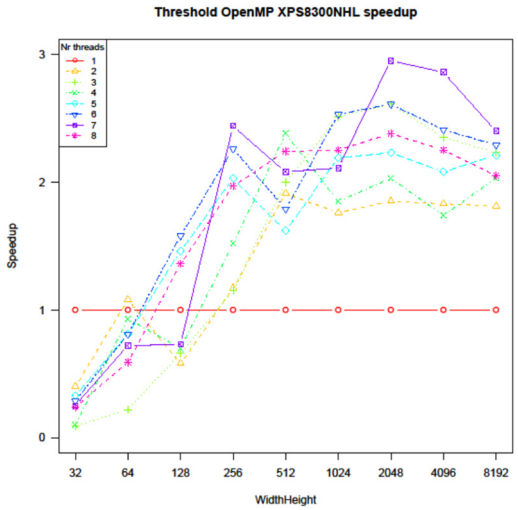
Sequential Threshold

```
template <class OrdImageT, class PixelT>
void Threshold (OrdImageT &image, const PixelT low, const PixelT high) {
   PixelT *pixelTab = image.GetFirstPixelPtr();
   const 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
```


Parallel 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 for
   for (int i = 0; i < nrPixels; i++) {
      his[pixelTab[i]]++;
   } // for i
} // Histogram0
```

OpenMP Histogram

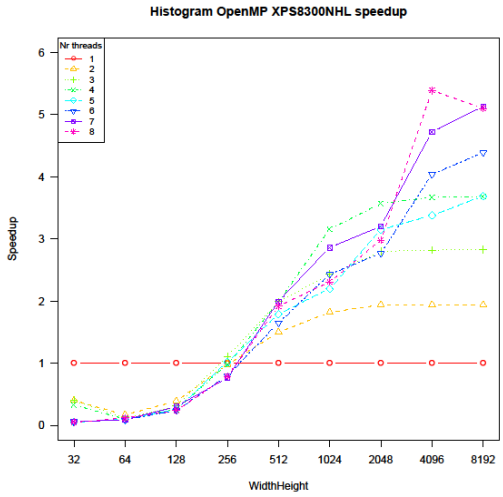
Pseudo code

- **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

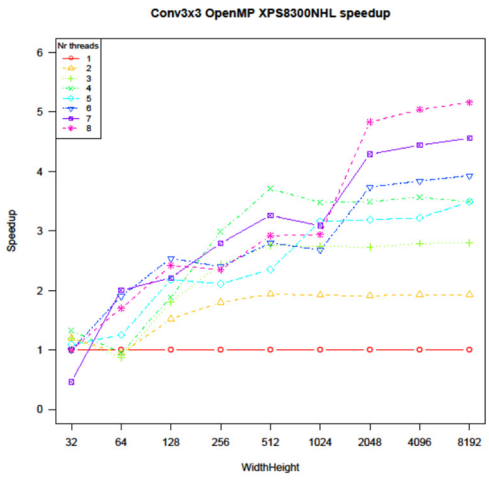
```
template <class IntImageT>
void Histogram0 (const IntImageT &image, const int hisSize, int *his) {
   typedef typename IntImageT::PixelType PixelT;
   memset(his, 0, hisSize * sizeof(int));
   PixelT *pixelTab = image.GetFirstPixelPtr();
   const int nrPixels = image.GetNrPixels();
#pragma omp parallel
    {
      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)



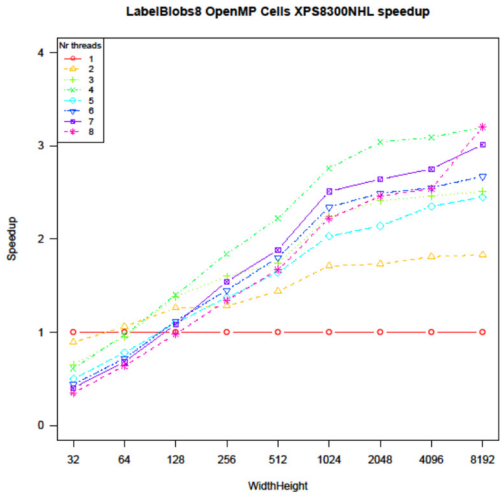
27-8-2018 OpenMP and OpenCL 59

OpenMP Convolution speedup graph on i7 2600 (4 cores)



27-8-2018 OpenMP and OpenCL 60

OpenMP LabelBlobs speedup graph on i7 2600 (4 cores)



27-8-2018 OpenMP and OpenCL 61

OpenCL

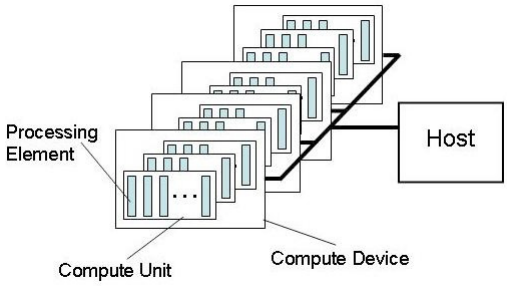
- **OpenCL architecture**
  - **Platform model**
  - **Execution model**
  - **Memory model**
  - **Programming model**
- **Kernel language**
- **Host API**
- **Examples**
- **Memory transfer ***

See for standard: www.khronos.org/opencl

27-8-2018 OpenMP and OpenCL 62

OpenCL platform model

The model consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more compute units which are further divided into one or more processing elements. Computations on a device occur within the processing elements.



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

27-8-2018 OpenMP and OpenCL 63

OpenCL execution model

Execution of an OpenCL program occurs in two parts:

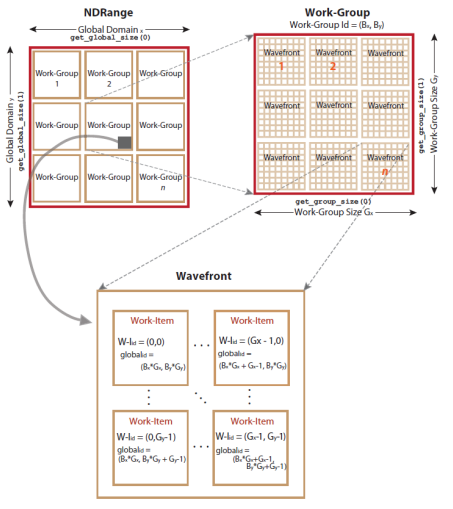
- **Kernels that execute on one or more OpenCL devices**
- **A host program that executes on the host**

27-8-2018 OpenMP and OpenCL 64

OpenCL execution model

When the kernel is submitted to the compute device for computation an indexing space is defined

NDRange, Work-Groups and Wavefronts



After AMD Accelerated Parallel Processing
OpenCL Programming GuideAMD, 2011

OpenCL execution model

When the kernel is submitted to the compute device for computation an indexing space is defined

An instance of the kernel, called work-item is created for each index

- **Work-item has a unique global ID**
- **Work-items are organized into work-groups**
- **All work-items of one work-group execute concurrently on the processing elements of a single compute unit**
- **Work-group has a unique work-group ID**
- **Work-item has a unique local ID within a work-group**

The indexing space is called NDRange (N-Dimensional Range)
OpenCL supports up to and including three dimensional indexing

OpenCL execution model

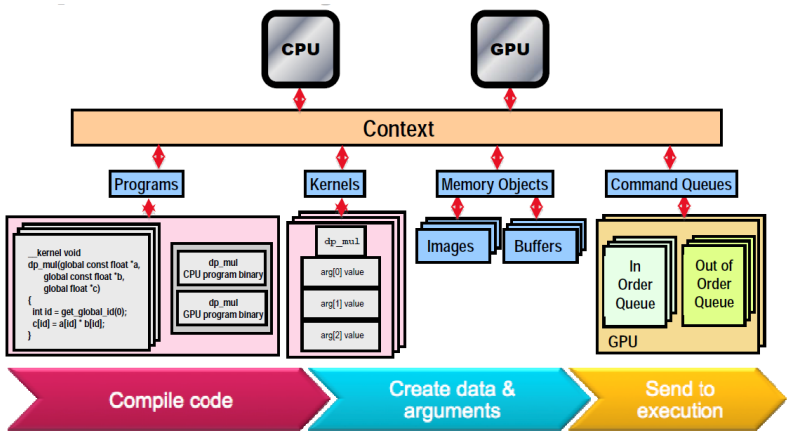
The host program defines the context for the kernels and manages their execution. The context includes:

- **Devices**
- **Program objects (source and executables)**
- **Memory objects (buffers, images, queues and events)**
- **Kernels (OpenCL functions that run on devices)**

The host places commands into the command-queue which are then scheduled onto the devices within the context:

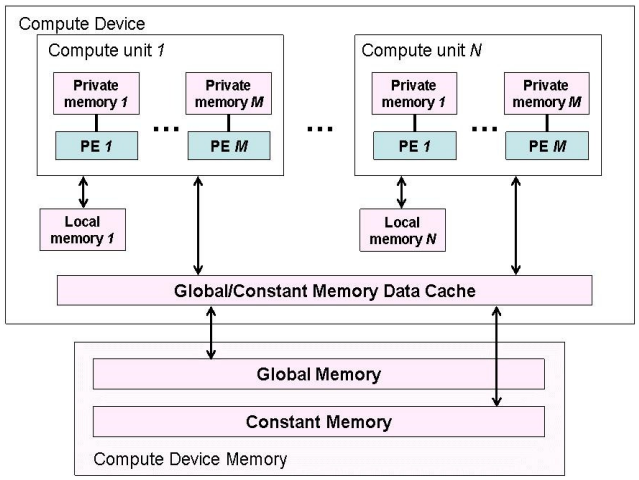
- **Kernel execution commands: execute a kernel on the processing elements of a device**
- **Memory commands: transfer data to, from, or between memory objects**
- **Synchronization commands: constrain the order of execution of commands**

OpenCL execution model



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

OpenCL memory model



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

OpenCL memory model

- **Global Memory: permits read/write access to all work-items in all work-groups**
- **Constant Memory: global memory initialized by host that remains constant during the execution of a kernel**
- **Local Memory: memory local to a work-group, variables are shared by all work-items in that work-group**
- **Private Memory: memory private to a work-item**

OpenCL programming model

Programming models

- **Supports data parallel and task parallel**
- **Primary model driving the design is data parallel**

Synchronization

- **In kernels**
  - **Work-items in a single work-group using a barrier**
  - **No mechanism for synchronization between work-groups**
- **Using host API**
  - **Command-queue barrier, commands enqueued to command-queue(s) in a single context**
  - **Using events**

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**
- **Announced OpenCL 2.1: subset of C++14**

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];
}
```

27-8-2018 OpenMP and OpenCL 73

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 initialize 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**

27-8-2018 OpenMP and OpenCL 74

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
```

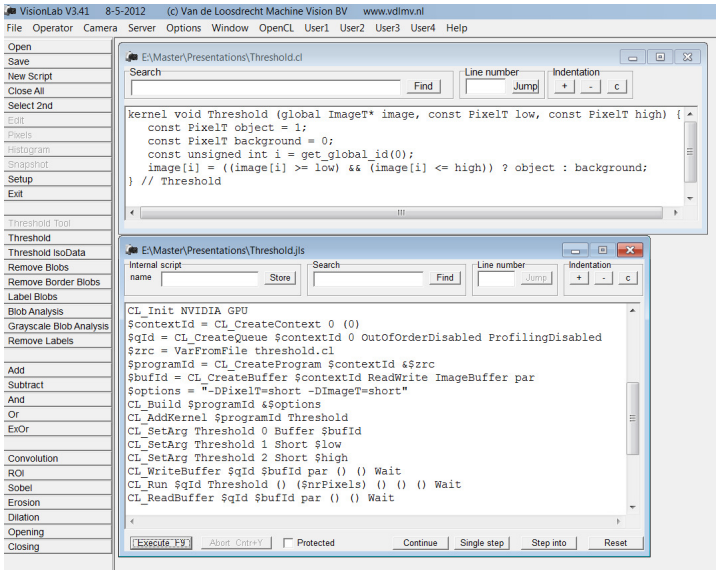
"Poor man's" template for Int16Image:

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

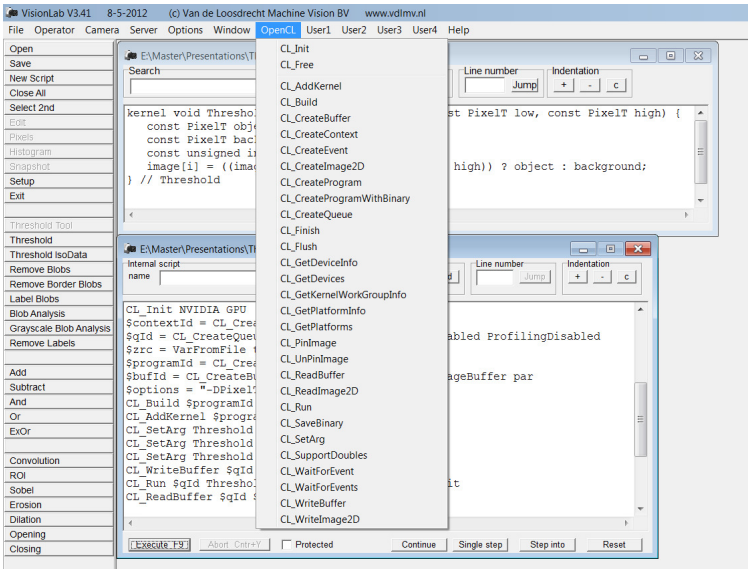
OpenCL Threshold host side VisionLab script

```
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
```

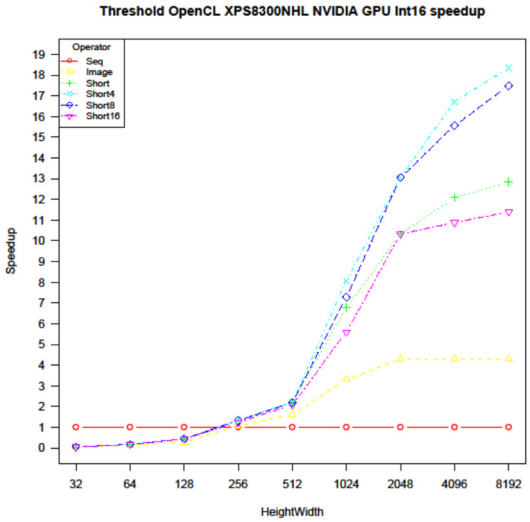
OpenCL development in VisionLab



OpenCL development in VisionLab

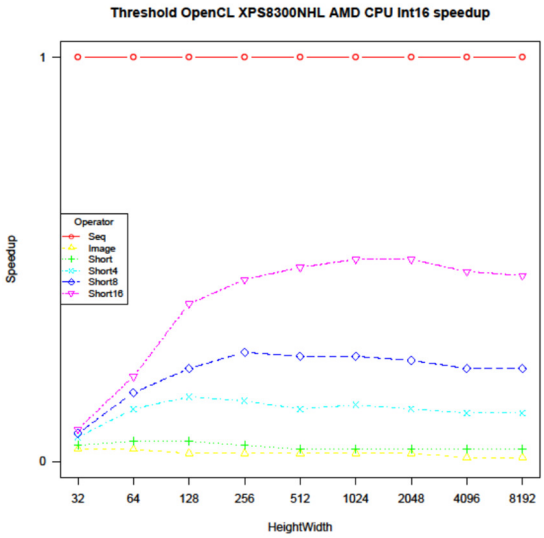


OpenCL Threshold GPU speedup graph (GTX 560 Ti)
One pixel or vector of pixels per kernel



27-8-2018 OpenMP and OpenCL 79

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



27-8-2018 OpenMP and OpenCL 80

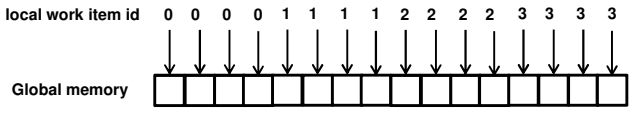
OpenCL Threshold kernel
Chunk of pixels or vectors of pixels per kernel

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

27-8-2018 OpenMP and OpenCL 81

Non-coalesced memory access of global memory

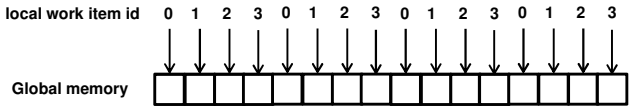
Example for warp with four work-items and chunk size = 4



27-8-2018 OpenMP and OpenCL 82

Coalesced memory access of global memory

Example for warp with four work-items and chunk size = 4



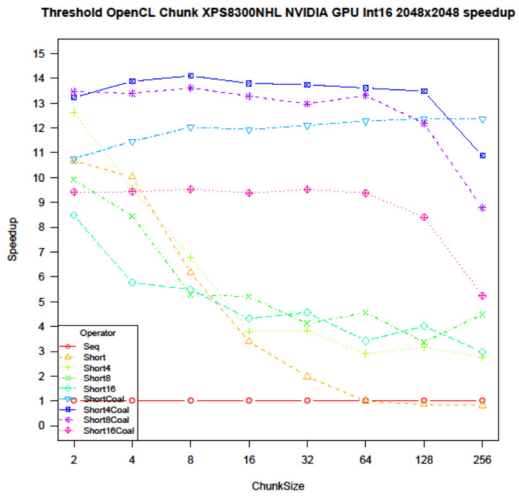
27-8-2018 OpenMP and OpenCL 83

OpenCL Threshold kernel
One pixel or vector of pixels per kernel with coalesced access

```
kernel void ThresholdCoalChunk (global ImageT* image,
               const PixelT low, const PixelT high,
               const unsigned int size) {
   const PixelT object = 1;
   const PixelT background = 0;
   const unsigned int gid = get_group_id(0);
   const unsigned int lid = get_local_id(0);
   const unsigned int ws = get_local_size(0);
   unsigned int i = (gid * ws * size) + lid;
   const unsigned int last = i + size * ws;
   for (; i < last; i += ws) {
     image[i] = ((image[i] >= low) && (image[i] <= high)) ?
                  object : background;
   } // for i
} // ThresholdCoalChunk
```

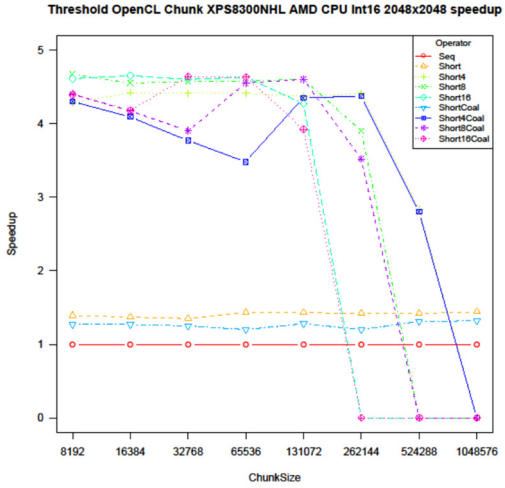
27-8-2018 OpenMP and OpenCL 84

OpenCL Threshold GPU speedup graph (GTX 560 Ti)
Chunk of pixels or vectors of pixels per kernel



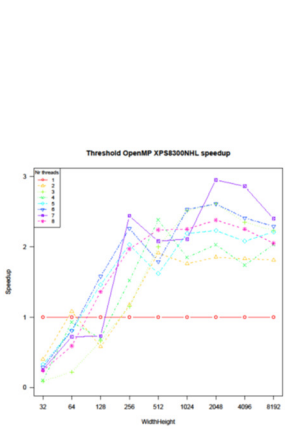
27-8-2018 OpenMP and OpenCL 85

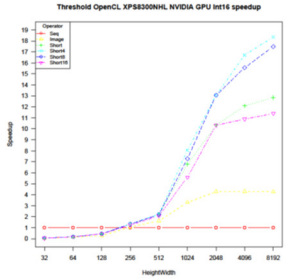
OpenCL Threshold CPU speedup graph (i7 2600)
Chunk of pixels or vectors of pixels per kernel

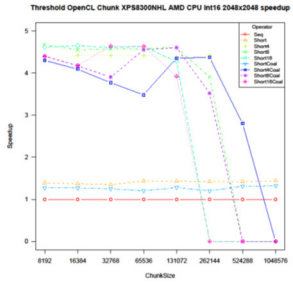


27-8-2018 OpenMP and OpenCL 86

Threshold OpenMP versus OpenCL

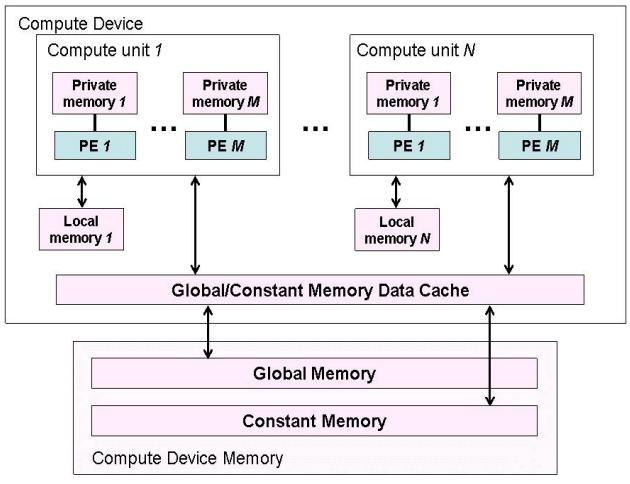






27-8-2018 OpenMP and OpenCL 87

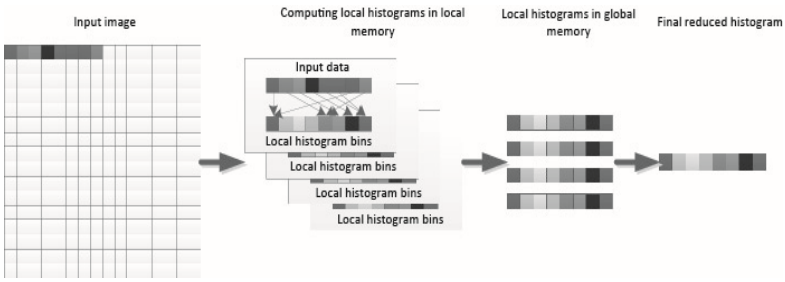
Device architecture (OpenCL)



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

27-8-2018 OpenMP and OpenCL 88

OpenCL Histogram



After Gaster et al., 2012, chapter 9

27-8-2018 OpenMP and OpenCL 89

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);
```

27-8-2018 OpenMP and OpenCL 90

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
```

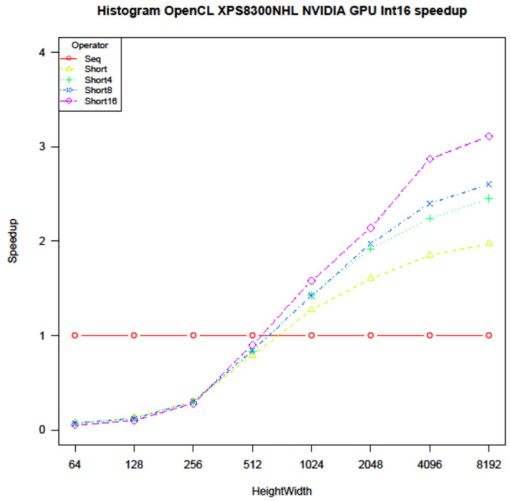
27-8-2018 OpenMP and OpenCL 91

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
```

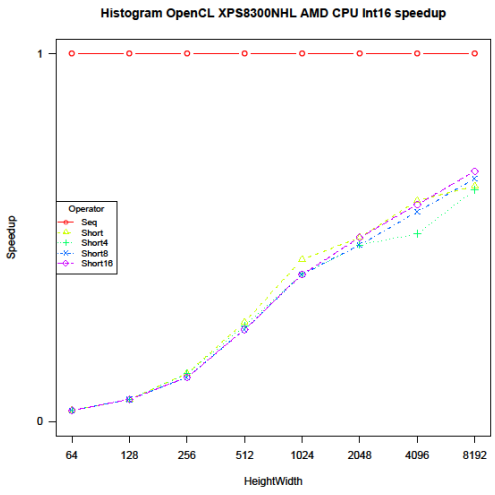
27-8-2018 OpenMP and OpenCL 92

OpenCL Histogram speedup graph(GTX 560 Ti)



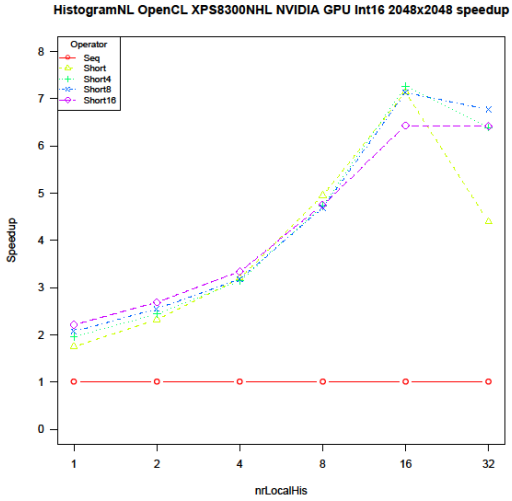
27-8-2018 OpenMP and OpenCL 93

OpenCL Histogram speedup graph(i7 2600)



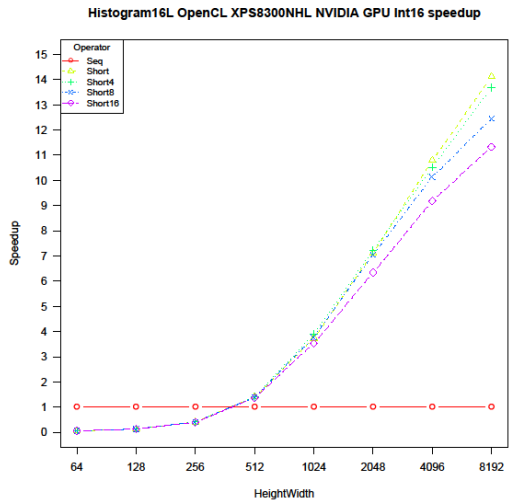
27-8-2018 OpenMP and OpenCL 94

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



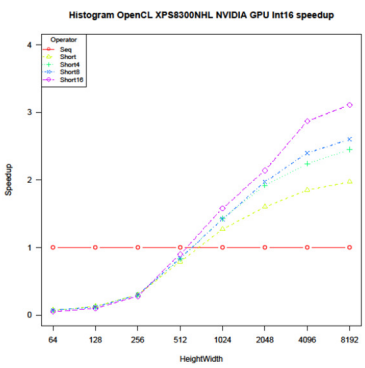
27-8-2018 OpenMP and OpenCL 95

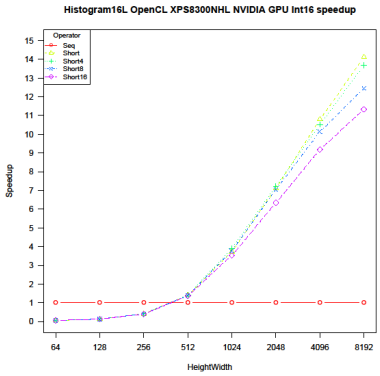
OpenCL Histogram GPU speedup graph (GTX 560 Ti)
with 16 local histograms



27-8-2018 OpenMP and OpenCL 96

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





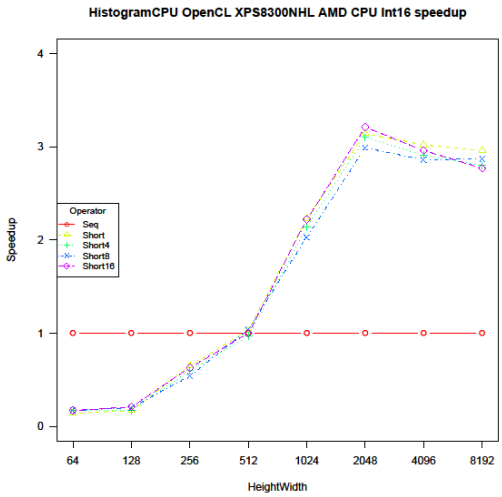
27-8-2018 OpenMP and OpenCL 97

Optimized implementation for CPUs

- **Each workgroup has only one work-item**
- **Number of workgroups is equal to the number of cores**
- **No race conditions for the local histogram, so no need for expensive atomic increment operations**

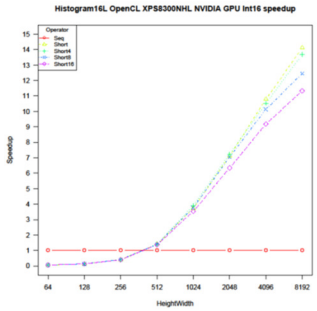
27-8-2018 OpenMP and OpenCL 98

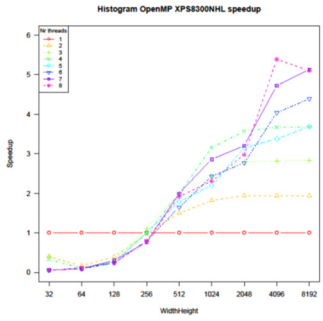
OpenCL Histogram optimized for CPU (i7 2600)

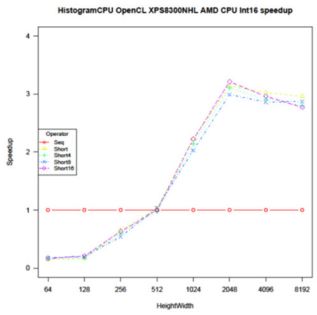


27-8-2018 OpenMP and OpenCL 99

Histogram OpenCL versus OpenMP



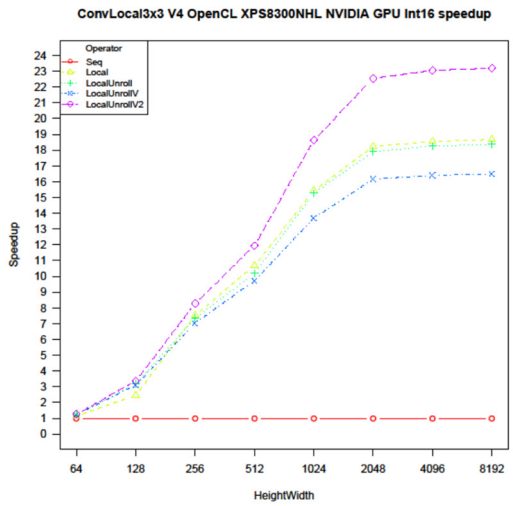




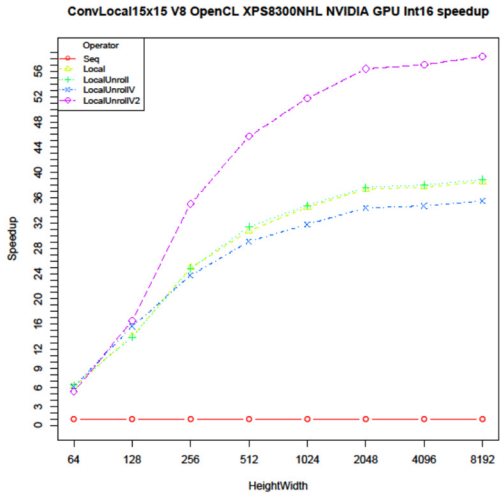
HistogramCPU OpenCL XPS8300NHL AMD CPU Int16 best work group

HeightWidth	Seq	Short	Short4	Short8	Short16
64	1	1	1	1	1
128	1	1	1	1	1
256	1	1	1	1	1
512	1	2	2	4	2
1024	1	4	8	8	8
2048	1	8	8	8	8
4096	1	8	8	8	8
8192	1	4	4	4	4

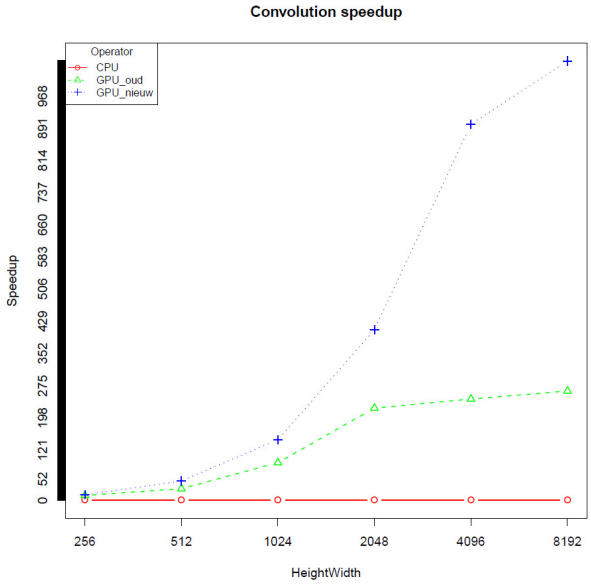
OpenCL Convolution GPU speedup graph (GTX 560 Ti)



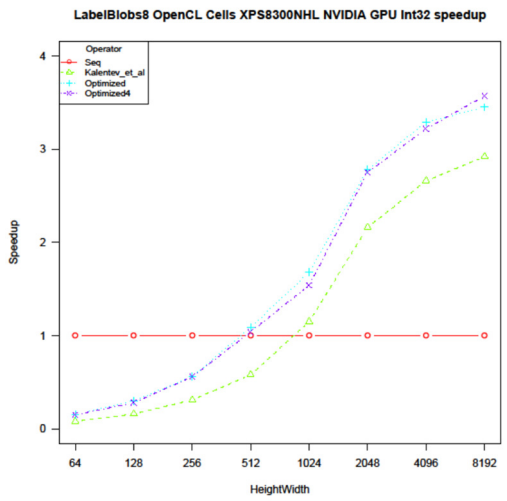
OpenCL Convolution GPU speedup graph (GTX 560 Ti)



Convolution 9x9: AMD FX9590 CPU (1 core) vs AMD R9 290X GPU



OpenCL LabelBlobs GPU speedup graph (GTX 560 Ti)



Memory transfer

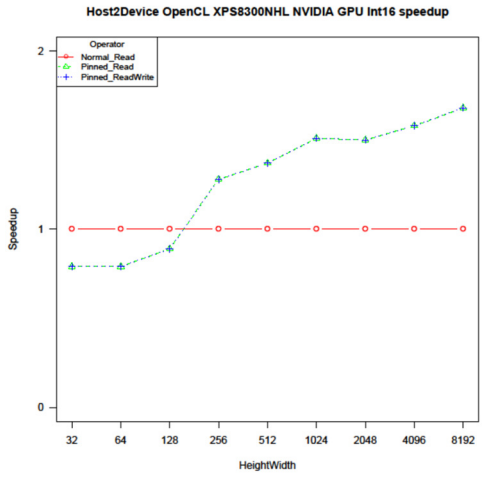
Memory transfers:

- **Normal**
- **Pinned**
- **Zero copy (not available for testing)**

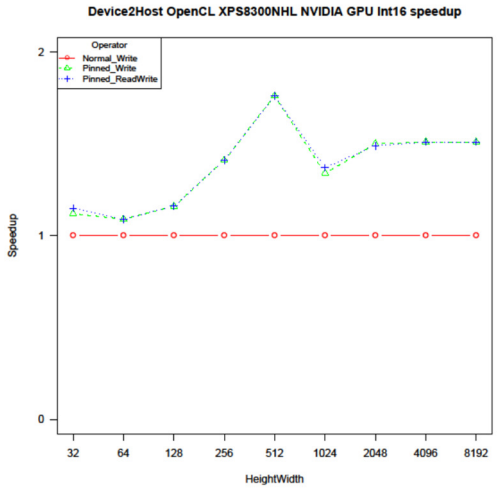
Data transfers benchmarked:

- **From CPU to GPU**
- **From GPU to CPU**

Data transfer from CPU to GPU (GTX 560 Ti)



Data transfer from GPU to CPU (GTX 560 Ti)



Overhead data transfer (in ms) is massive for simple vision operators

Host2Device OpenCL XPS8300NHL NVIDIA GPU Int16 median

HeightWidth	Normal_Read	Pinned_Read	Pinned_ReadWrite
32	26	33	33
64	26	33	33
128	33	37	37
256	68	53	53
512	179	131	131
1024	556	368	367
2048	1994	1329	1330
4096	8132	5146	5145
8192	34024	20298	20284

Threshold OpenCL XPS8300NHL NVIDIA GPU Int16 median

HeightWidth	Seq	Image	Short	Short4	Short8	Short16
32	2	48	41	41	41	41
64	7	47	41	41	41	42
128	18	67	40	40	41	41
256	57	54	43	43	43	46
512	202	125	92	91	92	97
1024	802	243	118	100	110	143
2048	3018	703	292	231	231	292
4096	11136	2586	921	666	715	1024
8192	43201	10110	3364	2354	2471	3790

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**
- **Tools are "in development"**
- **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 development 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**
- **Benchmark environment**
- **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**