*cl*OpenCL - Supporting Distributed Heterogeneous Computing in HPC Clusters

Albano Alves[1], José Rufino[1], António Pina[2], Luís Santos[2]

[1] Polytechnic Institute of Bragança, Portugal
[2] University of Minho, Portugal

HeteroPar'2012
August 27, 2012 - Rhodes, Greece

Contents

Heterogeneous Computing

Heterogeneous Computing

- combines different computing devices architectures (multi/many-core CPUs, GPGPUs, FPGAs, SoCs, ...) into an integrated execution environment ...
- ... to leverage the performance of applications, by exploiting the best capabilities of each device.

Challenges

- diversity of exec. environments and programming models
- not all algorithms / applications suitable to the new models
- new / unfamiliar memory hierarchies on computing devices
- need for explicit data transfers to/from computing devices
- ...

Heterogeneous Computing in Clusters

Clusters nodes with GPUs often exploited by an hybrid approach:

- MPI to distribute the application across multiple cluster nodes
- CUDA/OpenCL to run *kernels* on GPU(s) at each node
- PThreads/OpenMP to exploit CPU parallelism in each node

Porting applications from *single-node-multi-GPU* to *multi-node-multi-GPU* platforms may be quite demanding.

Multi-node-multi-accelerator heterogeneous computing should be as "straightforward" as in single-node-multi-accelerator scenarios.

Heterogeneous Computing with OpenCL

OpenCL - Open Computing Language

- an open programming standard for heterogeneous computing
- a typical OpenCL application is C99 based and comprises
    - a *host* program
    - a set of routines (*kernels*) to run on compute *evices*
- the OpenCL specification defines
    - a language for *kernel* programming
    - an API for *host* ↔ *devices* data transfers and *kernels* execution
- three major implementations, from different *vendors*
    - AMD APP SDK, for x86 CPUs and AMD GPUs
    - NVIDIA OpenCL SDK, for NVIDIA GPUs only
    - Intel OpenCL SDK, for x86 CPUs only

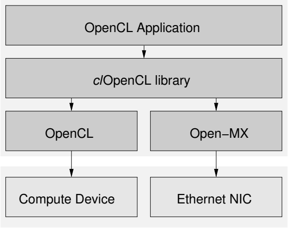
Heterogeneous Computing with OpenCL in Clusters

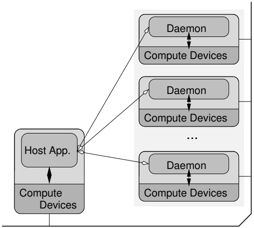
- OpenCL applications may only use the *local* compute *devices* of the machine where the *host* application component runs.

> To be able to use *remote* computing *devices*,
> the original OpenCL model must be extended.

- MGP (Many GPUs Package)
  - on top of VCL (MOSIX-like); binaries only; TCP sockets
  - single-system-image: virtual node with all cluster GPUs
  - "runs the CPU part of the application in a single node"
- Hybrid OpenCL
  - integrates the network layer in FOXC OpenCL; uses RPCs
  - bridge program (service) per remote node; x86 CPUs only
- our approach: *cl*OpenCL
  - works on top of any "canonical" OpenCL platform and is able to use any device (CPU, GPU, ...) supported by the platform
  - wrapper client library + remote services; uses Open-MX

*cl*OpenCL Architecture





- *cl*OpenCL consists of a *wrapper library* and a *daemon*
- the library redirects OpenCL calls to the local OpenCL runtime, or to a *cl*OpenCL daemon for remote execution
- daemons are OpenCL programs that handle remote calls (*cl*OpenCL library requests) and interact with local devices
- network data exchanges use Open-MX, a user-level low latency message passing stack over generic Ethernet

*cl*OpenCL Distributed Operation

- running a *cl*OpenCL application requires the prior launching of *cl*OpenCL daemons in cluster nodes with devices to be used
  - daemons are user-specific, started/stopped by users or jobs
  - per-user daemons isolate the OpenCL runtime of different users
- when the OpenCL *host* application starts, the *cl*OpenCL library (which also wraps *main*) discovers all daemons
  1. locally querying (`omx_info`) the Open-MX mapper service allows to discover all cluster nodes with Open-MX support
  2. querying (`omx_endpoint_info`) all Open-MX nodes (local and remote), allows to discover all nodes with *cl*OpenCL daemons
  3. `UID`s are used to identify user-specific *cl*OpenCL daemons
- different users may exploit different device combinations (sharing or not particular devices)

*cl*OpenCL Management of OpenCL Object References

- the standard OpenCL API handles objects of many types
  - platform and device identifiers, contexts, command queues, buffers, images, programs, kernels, events, samples, ...
- OpenCL objects are pointers to complex data structures
- *cl*OpenCL doesn't expose OpenCL pointers, once they lack global uniqueness (daemons have private address spaces)
- the *cl*OpenCL library returns globally unique "fake pointers" and maps them to local (real) pointers at specific daemons

*cl*OpenCL Platform and Device Querying

- a typical OpenCL application starts by discovering which (local) vendor-specific *platforms* (OpenCL implementations) are available and which (local) compute *devices* do they target
- in *cl*OpenCL, platform querying returns all local platforms, followed by all remote platforms (node by node) available in the cluster nodes where the user spawned *cl*OpenCL daemons
- `clGetPlatformInfo` was extended with the new attribute `CL_PLATFORM_HOSTNAME`, to allow OpenCL applications to know the cluster node to which a *platform* belongs
  - makes possible to select *devices* in specific cluster nodes

c/OpenCL Daemons Operation

c/OpenCL daemons are OpenCL programs that handle requests:

- each daemon creates a pool of listener threads
- each listener thread waits for request messages, using the Open-MX tagging and masking mechanism
- the request packet encloses all data required for executing the OpenCL primitive
- during the execution, additional data may be exchanged for *read* and *write* operations
- at the end of the execution, results are sent to the remote client and the thread returns to the waiting stage
- the asynchronous execution of primitives is supported by a specific thread that handles completion state

Testbed Cluster

Hardware: 4 cluster nodes (`node-[0-3]`)

- Intel Q9650 CPU (3GHz quad-core, 12Mb L2 cache)
- 8Gb of RAM (non-ECC DDR3 1333MHz)
- SysKonnect SK-9871 NIC (PCI64, 1GBps Ethernet)
- NVIDIA GTX460 GPU (1Gb of GDDR5 RAM)
  - `node-0` with two GPUs

Software:

- OS: Linux ROCKS 5.4
- OpenCL platforms: AMD SDK 2.6, CUDA 4.1.28
- Open-MX 1.5.2 with `mtu` 9000

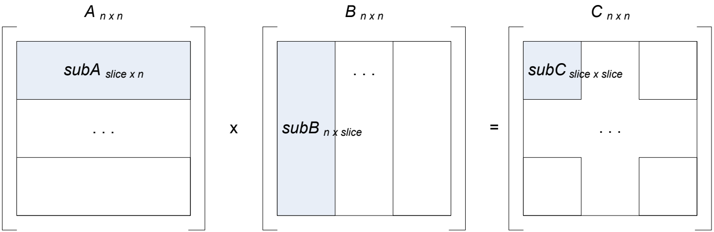
Test Application (1/5)

Matrix Product ($C = AB$)

- simple and "embarrassingly parallel"
    - check *cl*OpenCL correctness and scalability
    - HPC-class performance not our goal
- square matrices of order $n \in \{8K, 16K, 24K\}$
- single-precision elements (4 byte floats)
- size of each matrix: 256 Mbytes, 1 Gbyte, 2.25 Gbytes
    - 3×256 Mbytes $=$ 768 Mbytes $<$ 1 Gbyte of GPU RAM
    - 3×2.25 Gbytes $=$ 6.75 Gbytes $<$ 8 Gbytes of node RAM

Test Application (2/5)

Sliced Matrix Product



- A and B partitioned in sub-matrices $subA$ and $subB$
- C partitioned in sub-matrices $subC = subA \times subB$
- *slice*: height/width/order of $subA/subB/subC$

Test Application (3/5)

OpenCL kernel

```
_kernel void matrix_mult ( const int n, const int slice,
                           __global float * subA, __global float * subB,
                           __global float * subC ) {
    int i, j, k; float v=0;

    i = get_global_id(0); j = get_global_id(1);
    for(k=0; k<n; k++)
        v +=  subA[i*n+k] * subB[j*n+k];
     subC[i*slice+j] = v;
}
```

- important parameters of `clEnqueueNDRangeKernel`
  - `size_t global_work_size[2]` = {*slice*, *slice*}
    - to produce *subC* requires $slice^2$ work-items (kernel execs.)
  - `size_t local_work_size[2]` = {8,8}
    - hand-tuned; kernel doesn't take advantage of workgroups and so the definition of this parameter isn't straightforward

Test Application (4/5)

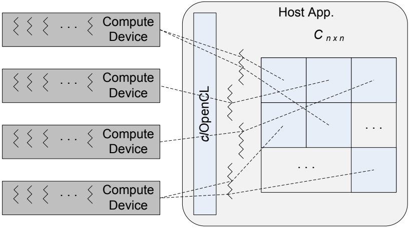
Slice definition

- simplification: same value of *slice* for all cluster devices
- $slice = 1K, 2K, 4K$ for $n = 8K, 16K, 24K$ (respectively)
  - in our cluster, a GPU is approx. twice as fast as a CPU
  - with 5 GPUs and 4 CPUs, we need at least $2 \times 5 + 4 = 14$ kernel executions to keep all the devices of the cluster busy
  - at least two kernel executions per device, for a more fine-grain load balancing $\Longrightarrow$ at least 28 kernel executions in total
  - num. of kernel execs. = num. of sub-matrices $subC = (n^2/slice^2)$

$$
\begin{aligned}
(n^2/slice^2) \geq 28 \quad & \Longrightarrow \quad slice \leq 1K \text{ for } n = 8K \quad (\Longrightarrow \geq 64 \text{ kernel execs.}) \\
& \Longrightarrow \quad slice \leq 2K \text{ for } n = 16K \quad (\Longrightarrow \geq 64 \text{ kernel execs.}) \\
& \Longrightarrow \quad slice \leq 4K \text{ for } n = 24K \quad (\Longrightarrow \geq 36 \text{ kernel execs.})
\end{aligned}
$$

Test Application (5/5)

Host component (Host App.)



- one thread (POSIX Threads) per OpenCL device
- per-thread dynamic work (auto-)assignment
  - i) select an unprocessed *subC*
  - ii) copy (*) *subA* and *subB* to device
    - (*) *subA* and *subB* reused when possible
  - iii) trigger the kernel execution
  - iv) collect and merge *subC* into *C*
  - v) go to step i)

Test Configurations

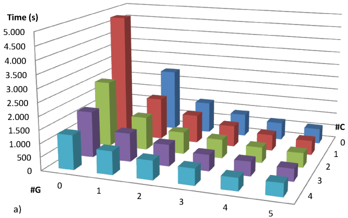
- *host* component executed on the cluster node with the most performant set of OpenCL devices: `node-0` (1CPU, 2 GPUs)
- overall, 74 combinations of devices where `node-0` is always used and zero or more remote nodes (`node-[1-3]`) are used
- most performant combinations (29) of CPUs (C) and GPUs (G), for a certain number of CPUs ($\#C$) and GPUs ($\#G$)

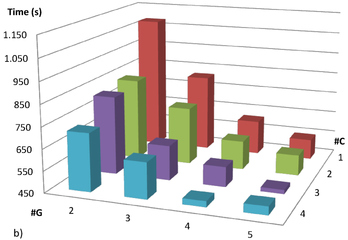
#C \ #G	0	1	2	3	4	5
0		G	GG	GG,G	GG,G,G	GG,G,G,G
1	C	GC	GGC	GGC,G	GGC,G,G	GGC,G,G,G
2	C,C	GC,C	GGC,C	GGC,G,C	GGC,G,G,C	GGC,GC,G,G
3	C,C,C	GC,C,C	GGC,C,C	GGC,G,C,C	GGC,GC,G,C	GGC,GC,GC,G
4	C,C,C,C	GC,C,C,C	GGC,C,C,C	GGC,GC,C,C	GGC,GC,GC,C	GGC,GC,GC,GC

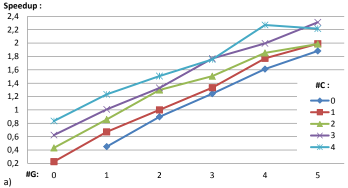
- use the maximum possible number of local (`node-0`) devices
- scatter as much as possible the remote (`node-[1-3]`) devices

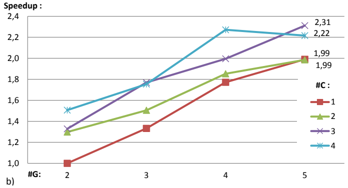
Test Results (1/6)

Execution Times and Speedups for $n = 24K$









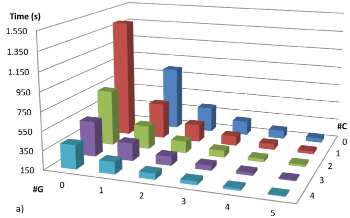
a) all (29) combinations

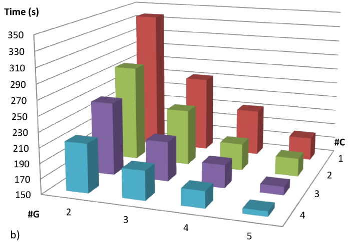
b) zoom: $\#C \geq 1$ and $\#G \geq 2$

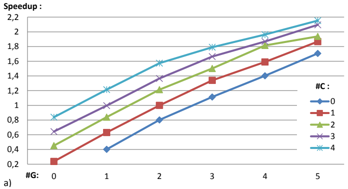
(notes: zoom graphics show *cl*OpenCL gains over OpenCL best scenario (GGC); speedup baseline (1,0) is GGC)

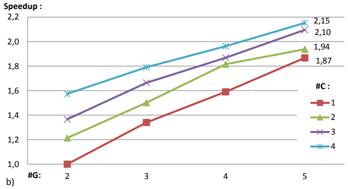
Test Results (2/6)

Execution Times and Speedups for $n = 16K$









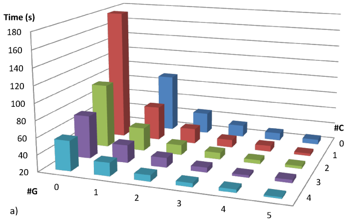
a) all (29) combinations

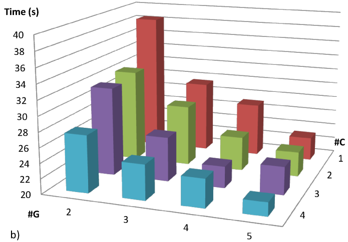
b) zoom: $\#C \geq 1$ and $\#G \geq 2$

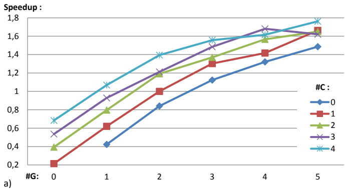
(notes: zoom graphics show *cl*OpenCL gains over OpenCL best scenario (GGC); speedup baseline (1,0) is GGC)

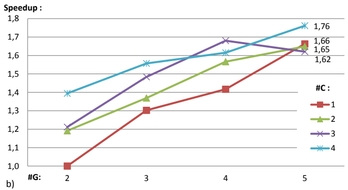
Test Results (3/6)

Execution Times and Speedups for $n = 8K$









a) all (29) combinations

b) zoom: $\#C \geq 1$ and $\#G \geq 2$

(notes: zoom graphics show *cl*OpenCL gains over OpenCL best scenario (GGC); speedup baseline (1,0) is GGC)

Test Results (4/6)

- exec. time decreases / speedup increases, with more devices
  - as expected, this trend is stronger when adding GPUs
  - too many devices may turn out to be counter-productive
- summary:

n	**Worst OpenCL**				**Best OpenCL**		**Best clOpenCL**		
	combination		*time*		*combination*	*time*	*combination*	*time*	*speedup*
24K	C	G	4800s	2403s	**GGC**	1084s	**GGC**,GC,GC,G	469s	2.31
16K	C	G	1422s	840s	**GGC**	337s	**GGC**,GC,GC,GC	157s	2.15
8K	C	G	179s	91s	**GGC**	38s	**GGC**,GC,GC,GC	22s	1.76

  - a single CPU takes approx. twice the time of a single GPU
  - *cl*OpenCL best scenarios
    - build on the OpenCL's best (**GGC**)
    - tend to maximize the device usage
  - speedup grows with n (scalability grows with the problem size)

Test Results (5/6)

- *cl*OpenCL speedup over OpenCL optimum (GGC) seems modest ...
- "how close are real (measured) speedups from ideal speedups" ?
- a GPU executes twice the kernels of a CPU in the same time
- for a single ideal cluster node, with $\#C$ CPUs and $\#G$ GPUs, the ideal (maximum) speedup would be $S_{ideal} = \#C + 2 \times \#G$
- comparing OpenCL optimum (GGC) with OpenCL worst case (C):

n	S_{real} [OpenCL]	S_{ideal} [OpenCL]	$\frac{S_{real}[OpenCL]}{S_{ideal}[OpenCL]}$
24K	S_{real}(GGC;C)=4.43	S_{ideal}(GGC;C)=5	88.6%
16K	S_{real}(GGC;C)=4.22	S_{ideal}(GGC;C)=5	84.4%
8K	S_{real}(GGC;C)=4.71	S_{ideal}(GGC;C)=5	94.2%

– $S_{real}(X;Y) = T(Y)/T(X)$ is the speedup of scenario X over Y
– T(Z) is the exec. time of device combination Z

Test Results (6/6)

- comparing *cl*OpenCL optimums with OpenCL worst case (C):

n	S_{real} [clOpenCL]	S_{ideal} [clOpenCL]	$\frac{S_{real}[clOpenCL]}{S_{ideal}[clOpenCL]}$
24K	S_{real}(GGC,GC,GC,G;C)=10.23	S_{ideal}(GGC,GC,GC,G;C)=13	78.7%
16K	S_{real}(GGC,GC,GC,GC;C)=9.05	S_{ideal}(GGC,GC,GC,GC;C)=14	64.7%
8K	S_{real}(GGC,GC,GC,G;C)=8.21	S_{ideal}(GGC,GC,GC,GC;C)=14	58.7%

– $S_{real}(X;Y) = T(Y)/T(X)$ is the speedup of scenario X over Y
– T(Z) is the exec. time of device combination Z

- comparing *cl*OpenCL optimums with OpenCL optimum (GGC):

n	$\alpha = \frac{S_{real}[clOpenCL]}{S_{real}[OpenCL]}$	$\beta = \frac{S_{ideal}[clOpenCL]}{S_{ideal}[OpenCL]}$	$\frac{\alpha}{\beta}$
24K	10.23 / 4.43 = 2.31 (*)	13 / 5 = 2.6	**88.8%**
16K	9.05 / 4.22 = 2.14 (*)	14 / 5 = 2.8	**76.4%**
8K	8.21 / 4.71 = 1.74 (*)	14 / 5 = 2.8	**62.1%**

– α (β) = real (ideal) speedup of *cl*OpenCL over OpenCL
– (*) ≈ speedup values of slide 23

Conclusions and Future Work

- *c*/OpenCL enables the execution of cluster-wide OpenCL applications in commodity HW and without special privileges
- porting OpenCL applications to *c*/OpenCL is straightforward
  - no source changes; just link with *c*/OpenCL + Open-MX libs
- benchmark results show fair performance and good scalability
  - also allowed to identify different combinations of devices with the same performance level, which may be used in alternative
- future work
  - expand the number of OpenCL primitives supported
  - conformance/performance tests (Rodinia, Vienna CL, SHOC)
  - BSD sockets support (less performance, better portability)
    - ongoing work (almost complete)
- source code: ongoing work (available on request)

Thank you ! Questions ? Remarks ?