MANY-CORE COMPUTING

Ana Lucia Varbanescu, UvA

Original slides: Rob van Nieuwpoort, eScience Center

1-Oct-2015
Schedule

1. Introduction and Programming Basics (24-9-2015)
3. Advanced GPU programming (1-10-2015)
4. Heterogeneous programming (5-10-2015)
Today

- Heterogeneous computing
  - And why do we care?
- Programming heterogeneous systems
  - Programming models
- Dealing with applications
  - Static vs. Dynamic scheduling
  - Imbalanced applications
  - Multi-kernel applications
  - Run-time based systems
Goals

- Get you interested in heterogeneous computing.
- Introduce methods for efficient heterogeneous computing
  - Single-node
  - Large-scale
- Practical examples
- Current challenges and open research questions.
- Fair to others, but advertise our research 😊
Reminder

Collaborate for the same workload.
Collaborate?

☐ Why?
   - Gain performance
     - Different workload parts for different processors
     - Availability
     - Data communication bottleneck

☐ When?
   - Not all applications work best on a GPU.

☐ How?
   - Must address communication.
   - Must tackle partitioning.
Programming Heterogeneous Systems
Heterogeneous programming?

- Mixes different languages for different “kernels”
  - OpenMP + CUDA
  - TBB + CUDA
  - ...
- Single source => same language (+ pragma’s)
  - OpenCL (from Khronos)
  - OpenACC (from NVIDIA, PGI, …)
  - HPL (from University of A Coruna, Spain)
  - OmpSS (from UPC, Spain)
  - ...
- Specific APIs and libraries
Programming models

- Vendor-specific: e.g., NVIDIA CUDA
  - Portability concerns only legacy code
  - (Productivity is increased by hiding more and more details of the machine, assumed known)

- “Community” programming models
  - High-level: OpenMP, OpenACC, OmpSS, …
  - Not-so-high-level: OpenCL (may call it low, but is unfair)
  - (Very-)Low-level: assembly, C, intrinsics, …
Single-source ↔ portability

- **Portability**: the ability of the same implementation of an application to run unchanged on different platforms

- Inter-family vs inter-vendor
  - NVIDIA Cuda runs on all NVIDIA GPU families
  - OpenCL runs on all GPUs, Cell, CPUs

- Parallelism portability
  - Different architecture require different granularity
  - Task vs data parallel

- Performance portability
  - Can we express platform-specific optimizations?
An example: OpenCL

- **Open standard** for **portable** multi-core programming
  - Supported by the KHRONOS group
- **Architecture independent**
  - Explicit support for multi-/many-cores
- **Low-level host API**
  - High-level bindings (e.g., Java, Python)
- **Separate high-level kernel language**
  - Explicit support for vectorization
- **Run-time compilation**
- **Architecture-dependent optimizations**
  - Required, explicitly implemented
OpenCL platform model

- Work-item
- Work-group
- Compute kernels
- A host program
OpenCL Memory model
Kernels are the main functional units in OpenCL
- Kernels are executed by work-items
- Work-items are mapped transparently on the hardware platform

Functional portability is guaranteed
- Programs run correctly on different families of hardware
- Explicit platform-specific optimizations are dangerous

Performance portability is another discussion
- Ask me about it!
# Cuda vs OpenCL Terminology

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Work item</td>
</tr>
<tr>
<td>Thread block</td>
<td>Work group</td>
</tr>
<tr>
<td>Device memory</td>
<td>Global memory</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Constant memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Local memory</td>
</tr>
<tr>
<td>Local memory</td>
<td>Private memory</td>
</tr>
</tbody>
</table>
Cuda vs OpenCL Qualifiers

### Functions

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong></td>
<td>__kernel</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>(no qualifier needed)</td>
</tr>
</tbody>
</table>

### Variables

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>constant</strong></td>
<td><strong>constant</strong></td>
</tr>
<tr>
<td><strong>device</strong></td>
<td><strong>global</strong></td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td><strong>local</strong></td>
</tr>
</tbody>
</table>
Cuda vs OpenCL Indexing

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridDim</td>
<td>get_num_groups()</td>
</tr>
<tr>
<td>blockDim</td>
<td>get_local_size()</td>
</tr>
<tr>
<td>blockIdx</td>
<td>get_group_id()</td>
</tr>
<tr>
<td>threadIdx</td>
<td>get_local_id()</td>
</tr>
<tr>
<td>Calculate manually</td>
<td>get_global_id()</td>
</tr>
<tr>
<td>Calculate manually</td>
<td>get_global_size()</td>
</tr>
</tbody>
</table>

`__syncthreads()` → `barrier()`
Vector add: Cuda vs OpenCL kernel

CUDA

```c
__global__ void
vectorAdd(float* a, float* b, float* c) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

OpenCL

```c
__kernel void
vectorAdd(__global float* a, __global float* b, __global float* c) {
    int index = get_global_id(0);
    c[index] = a[index] + b[index];
}
```
const size_t workGroupSize = 256;
const size_t nrWorkGroups = 3;
const size_t totalSize = nrWorkGroups * workGroupSize;

cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

// create properties list of key/values, 0-terminated.
cl_context_properties props[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0
};

cl_context context = clCreateContextFromType(props,
    CL_DEVICE_TYPE_GPU, 0, 0, 0);
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1,
       &device, NULL);

// create command queue on 1st device the context reported
cl_command_queue commandQueue =
    clCreateCommandQueue(context, device, 0, 0);

// create & compile program
cl_program program = clCreateProgramWithSource(context, 1,
       &programSource, 0, 0);
clBuildProgram(program, 0, 0, 0, 0, 0, 0, 0);

// create kernel
cl_kernel kernel = clCreateKernel(program, "vectorAdd",0);
float* A, B, C = new float[totalSize]; // alloc host vecs
// initialize host memory here...

// allocate device memory
cl_mem deviceA = clCreateBuffer(context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    totalSize * sizeof(cl_float), A, 0);

cl_mem deviceB = clCreateBuffer(context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    totalSize * sizeof(cl_float), B, 0);

cl_mem deviceC = clCreateBuffer(context,
    CL_MEM_WRITE_ONLY, totalSize * sizeof(cl_float), 0, 0);
// setup parameter values
clSetKernelArg(kernel, 0, sizeof(cl_mem), &deviceA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &deviceB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &deviceC);

clEnqueueNDRangeKernel(commandQueue, kernel, 1, 0,
    &totalSize, &workGroupSize, 0, 0, 0); // execute kernel

// copy results from device back to host, blocking
clEnqueueReadBuffer(commandQueue, deviceC, CL_TRUE, 0,
    totalSize * sizeof(cl_float), C, 0, 0, 0);

delete[] A, B, C; // cleanup
clReleaseMemObject(deviceA); clReleaseMemObject(deviceB);
clReleaseMemObject(deviceC);
Main challenge

- Performance portability
  - Definition
  - Metrics
  - Goals
- At the moment, OpenCL is NOT performance portable.
- Very few models try to be:
  - High-level front-end => code specialization + high performance
  - Restricted language => low performance
Portability challenges

- Memory access patterns
  - Different for CPU and GPU

- Use of local memory
  - Local memory on CPUs ?!

- Device-to-host transfers
  - CPU has one memory space...

- Parallelism granularity
  - Few cores vs. thousands of cores

- Compilers
  - Still under development
High-level models

- An example: HPL
  - (i.e., Heterogeneous Programming Library)
  - High-level API, architecture independent
  - Different back-ends for different devices

```c
void Add(Array<float,1> a, Array<float,1> b,
         Array<float,1> c) {
    if (Idx < c.size) c[Idx] = a[Idx] + b[Idx];
}

#define N 100000
int main (…)
    Array<float, 1> a(N), b(N), c(N);
    eval(Add).device(GPU).global(1000).local(10) (a, b, c);
```
Heterogeneous computing

Main challenges
We assume ...

- A heterogeneous system: CPU + GPU
- Existing CPU code
  - OpenCL, OpenMP, HPL, OpenACC, ...
- Existing GPU code
  - OpenCL, CUDA, HPL, OpenACC, ...
- An application = algorithm + data
Using the platform effectively

Consider:

- Application parallelism
- CPU-GPU data transfers
- CPU can help processing
Example 1

- Dot product
  - Compute the dot product of 2 (1D) arrays

- GPU-friendly?

- CPU-friendly?
Example 1: dot product

<table>
<thead>
<tr>
<th>Execution time (ms)</th>
<th>TG</th>
<th>TD</th>
<th>TC</th>
<th>TMax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(100,0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(90,10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(80,20)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(70,30)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(60,50)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(50,50)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(40,60)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(30,70)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(20,80)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(10,90)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,100)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 2

- Separable convolution (CUDA SDK)
  - Apply a convolution filter (kernel) on a large image.
  - Separable kernel allows applying
    - Horizontal first
    - Vertical second

- GPU-friendly?
- CPU-friendly?
### Example 2: Separable convolution

<table>
<thead>
<tr>
<th>Executions (ms)</th>
<th>TG</th>
<th>TD</th>
<th>TC</th>
<th>TMax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(100,0)</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(90,10)</td>
<td></td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(80,20)</td>
<td></td>
<td></td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>(70,30)</td>
<td></td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>(60,50)</td>
<td></td>
<td></td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>(50,50)</td>
<td></td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>(40,60)</td>
<td></td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>(30,70)</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>(20,80)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>(10,90)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,100)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 3

- Matrix multiply
  - Multiply two dense matrices

- GPU-friendly?
- CPU-friendly?
So ...

- There are very few GPU-only applications
  - The CPU – GPU communication bottleneck.
  - The increasing performance of CPUs
- A part of the computation can always be done by the CPU.
  - Which part?

Main challenge:
How to partition the application.
Determining the partition

- Static partitioning (SP) vs. Dynamic partitioning (DP)
Static vs. dynamic

- **Static partitioning**
  - + can be computed before runtime => no overhead
  - + can detect GPU-only/CPU-only cases
  - + no unnecessary CPU-GPU data transfers
  - -- does not work for all applications

- **Dynamic partitioning**
  - + responds to runtime performance variability
  - + works for all applications
  - -- incurs (high) runtime scheduling overhead
  - -- might introduce (high) CPU-GPU data-transfer overhead
  - -- might not work for CPU-only/GPU-only cases
Static partitioning
Comparison of different methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy (best performance)</th>
<th>Speed</th>
<th>Pre-cost</th>
<th>Adaptive to app changes</th>
<th>Adaptive to SW, HW changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>++/--</td>
<td>++/--</td>
<td>n/a</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>TE</td>
<td>+++</td>
<td>---</td>
<td>n/a</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>AM</td>
<td>++/++</td>
<td>++/-</td>
<td>--/-</td>
<td>Y/N</td>
<td>Y/N</td>
</tr>
<tr>
<td>LM</td>
<td>++/++</td>
<td>+++</td>
<td>---</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- Manual (or arbitrary) mapping (MM)
- Try-and-error search (TE)
- Analytical modeling (AM)
- Learning-based method (LM)
The Glinda Framework

- A static-partitioning framework, based on auto-tuning
  - It targets massive data-parallel single-kernel applications
  - It aims to find the optimal partitioning, leading to the best performance
- Not the only one …
  - Qilin, SDMK
The Glinda Framework

App and parameters

1. User interface

2. Workload probe

3. HW profiler

HW capabilities

Repository

CPU+GPU

Partitioning

4. Matchmaker

5. Partitioner

Auto-tuner

6. Execution unit

Code library

What is wrong here?
Glinda’s Pros and Cons

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Finds the optimal partitioning, leading to the best performance</td>
<td>- Occupies the target platform to run the tuning</td>
</tr>
<tr>
<td>+ Utilizes the platform efficiently</td>
<td>- Time-consuming</td>
</tr>
<tr>
<td>+ Adapts to app and platform (SW, HW) changes</td>
<td></td>
</tr>
</tbody>
</table>
**Glinda’s Pros and Cons**

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Finds the optimal partitioning, leading to the best performance</td>
<td>- Occupies the target platform to run the tuning</td>
</tr>
<tr>
<td>+ Utilizes the platform efficiently</td>
<td>- Time-consuming</td>
</tr>
<tr>
<td>+ Adapts to app and platform (SW, HW) changes</td>
<td></td>
</tr>
</tbody>
</table>
We add AM into Glinda

<table>
<thead>
<tr>
<th></th>
<th>Accuracy (best performance)</th>
<th>Speed</th>
<th>Pre-cost</th>
<th>Adaptive to app changes</th>
<th>Adaptive to SW, HW changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>++/--</td>
<td>++/--</td>
<td>n/a</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>TE</td>
<td>+++</td>
<td>---</td>
<td>n/a</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>AM</td>
<td>+/++</td>
<td>++/-</td>
<td>--/-</td>
<td>Y/N</td>
<td>Y/N</td>
</tr>
<tr>
<td>LM</td>
<td>+/++</td>
<td>+++</td>
<td>---</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- Manual(or arbitrary) mapping (MM)
- Try-and-error search (TE)
- Analytical modeling (AM)
- Learning-based method (LM)

We want to optimize the partitioning process by predicting the partitioning
Our method*

- Model (quantify)
  - the app workload,
  - the HW capabilities,
  - the GPU-CPU data-transfer.

- Build the partitioning model
  - An equation to solve.

- Predict the partitioning

*Jie Shen et al., HPCC’14.
“Look before you Leap: Using the Right Hardware Resources to Accelerate Applications”
Model the app workload

n: the total problem size
w: workload per work-item
Model the app workload

$n$: the total problem size

$w$: workload per work-item

$W = w \times n \times \beta$

$W_G = w \times n \times \beta$

$W_C = w \times n \times (1 - \beta)$

$W$ (total workload size) quantifies how much work has to be done.
Hardware capabilities: $P = \text{processing throughput}$
- Measured as workload processed per second
- $P$ evaluates the hardware capability of a processor

GPU kernel execution time: $T_G = \frac{W_G}{P_G}$

CPU kernel execution time: $T_C = \frac{W_C}{P_C}$
Model the CPU-GPU data-transfer

- \( O \) = GPU data-transfer size
  - Measured in bytes
- \( Q \) = GPU data-transfer bandwidth
  - Measured in bytes per second

Data-transfer time: \( TD = \frac{O}{Q} + (\text{Latency}) \)

Latency < 0.1 ms, negligible impact
Three types of data transfers

- **ND**: $O=\text{zero}$ (no data transfer/data-transfer overhead can be ignored)
- **PD**: $O=\beta \times \text{Full data-transfer size (partial data transfer)}$
- **FD**: $O$ is independent of $\beta$ (full data transfer)

The data-transfer type specifies whether to take $T_D$ into account or not.
Build the partitioning model

Define the optimal partitioning

\[ T_G = T_C \]

\[ T_G + T_D = T_C \]
Build the partitioning model

- Substitute the quantities

**ND**

\[ T_G = T_C \]

\[ \frac{W_G}{P_G} = \frac{W_C}{P_C} \]

\[ \frac{W_G}{W_C} = \frac{P_G}{P_C} \]

The relative HW capability

**PD, FD**

\[ T_G + T_D = T_C \]

\[ \frac{W_G}{P_G} + \frac{O}{Q} = \frac{W_C}{P_C} \]

\[ \frac{W_G}{W_C} = \frac{P_G}{P_C} \times \frac{1}{1 + \frac{P_G}{Q} \times \frac{O}{W_G}} \]

The impact of data transfer
Predict the partitioning

- Solve the equation

\[
\frac{W_G}{W_C} = \frac{P_G}{P_C} \times \frac{1}{1 + \frac{P_G}{Q} \times \frac{O}{W_G}}
\]
Predict the partitioning

- Solve $\beta$ from the equation

**ND**

$$\beta = \frac{R_{GC}}{1 + R_{GC}}$$

$$O = O^V = 0$$

**PD**

$$\beta = \frac{R_{GC}}{1 + \frac{v}{w} \times R_{GD} + R_{GC}}$$

$$O = O^V \times \beta$$

**FD**

$$\beta = \frac{R_{GC} - \frac{v}{w} \times R_{GD}}{1 + R_{GC}}$$

$$O = O^V \neq 0$$
# Glinda’s Pros and Cons

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Find the optimal partitioning, leading to the best performance</td>
<td>- Occupy the target platform to run the tuning</td>
</tr>
<tr>
<td>- Utilize the platform efficiently</td>
<td>- Time-consuming</td>
</tr>
<tr>
<td>- Adapt to app and platform (SW, HW) changes</td>
<td></td>
</tr>
</tbody>
</table>

- Case study

What is still wrong here?
The Glinda Framework

1. User interface
2. Workload probe
3. HW capabilities
4. Matchmaker
5. Partitioner
6. Execution unit

Workload model

Repository

App and parameters

Hardware capabilities

CPU+GPU

Partitioning

Only-CPU/Only-GPU/CPU+GPU

Auto-tuner

Predictor

Code library
Experimental evaluation

- **Effectiveness (compared to Only-CPU/Only-GPU)**
  - Up to 12.6x/6.6x speedup
  - Only GPU: up to 96% performance will be lost
BONUS: Imbalanced applications

(a) Imbalanced workload

(b) Balanced workload
Sound ray tracing

- A collaboration with Dutch NLR
- Simulate the sound propagation
  - from an aircraft to receivers
- Assess aircraft flyover noise during the aircraft take-off and approach procedures
Sound ray tracing
Workload profile

(a) the original workload
Workload profile

(a) the original workload

(b) after sorting
Model

![Graph showing workload and ray ID](image)

- **Workload (# iterations)**
- **Ray ID (parallelization dimension)**

Sorting:

- **Flat**
- **Peak**
Modeling
Results [1]

The chart shows the execution time (s) for different configurations:
- **Only GPU**
- **Only CPU**
- **CPU+GPU (Predictor)**
- **CPU+GPU (Auto-tuner)**

The chart indicates a 62% performance improvement compared to “Only-GPU” for the W9 (1.3GB) workload.
Results on 10 workloads

- Changing the flight parameters we obtain 10 different workloads.
The more imbalance, the larger speedup!
Why use Glinda?

- Glinda enables the use of heterogeneous hardware for OpenCL workloads
  - Static partitioning is computed and implemented
  - Works for multi-GPUs
  - Works for data parallel kernels
- Resulting partitioned code can be re-used with minimum to moderate effort:
  - Different datasets
  - Different scenarios
  - Different platforms
How to use Glinda?

- Profile the platform: $R_{GC}$, $R_{GD}$
- Configure and use the solver: $\beta$
- Take the decision: Only-CPU, Only-GPU, CPU+GPU (and partitioning)
  - if needed, apply the partitioning

- Code preparation
  - Parallel implementations for both CPUs and GPUs
  - Enable profiling and partitioning

- Code reuse
  - Single-device code and multi-device code are reusable for different datasets and HW platforms
Glinda in HPL

- HPL = Heterogeneous Programming Library*

- Two usage scenarios:
  - on top of existing OpenCL kernels
  - developing the heterogeneous kernels in the embedded language it provides.

- Implementation of Glinda in HPL
  - Automatically profiles the application
  - Computes the partitioning
  - It applies it on the array

- Minimizes programmer effort.

* HPL: http://hpl.des.udc.es
Summary: static approaches

- Main goal: statically partition the workload once

- Main challenges:
  - Determine platform capabilities
  - Determine application performance characteristics
    - Profiling
      - On-line training, not feasible for one-time execution
    - Modeling
      - Off-line training + machine learning
      - On-line training + curve fitting

- Programming models: CUDA, OpenMP, TBB, OpenCL
  - Extra APIs provided
<table>
<thead>
<tr>
<th>Feature</th>
<th>Glinda</th>
<th>Qilin</th>
<th>Insieme</th>
<th>SKMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td>Online: Profiling per datasize</td>
<td>Online: Profiling per datasize + sampling</td>
<td>Special training set</td>
<td>On-line profiling</td>
</tr>
<tr>
<td>Partitioning</td>
<td>Off-line</td>
<td>On-line</td>
<td>On-line</td>
<td>On-line</td>
</tr>
<tr>
<td>Granularity</td>
<td>Data-driven</td>
<td>Data-driven</td>
<td>Data-driven</td>
<td>Workgroup</td>
</tr>
<tr>
<td>Model</td>
<td>Analytical</td>
<td>Curve-fitting</td>
<td>Machine learning</td>
<td>Decision Tree Heuristics</td>
</tr>
<tr>
<td>GPUs</td>
<td>Multi, asymmetrical</td>
<td>Single</td>
<td>Single [?]</td>
<td>Multi, asymmetrical</td>
</tr>
<tr>
<td>Programming model</td>
<td>OpenCL (OpenMP+CUDA)</td>
<td>TBB+CUDA</td>
<td>OpenCL</td>
<td>OpenCL</td>
</tr>
<tr>
<td>Manual effort</td>
<td>Enable multi-device code</td>
<td>Little</td>
<td>None</td>
<td>Little</td>
</tr>
</tbody>
</table>
Dynamic approaches
An example – single-node*

- Dynamic OpenCL-based approach to work-partitioning
  - no offline training
  - responds automatically to runtime performance variability

- Single kernel
  - partition the kernel efficiently into chunks of contiguous work groups
  - Schedule chunks for execution across multiple devices

Boyer et al, Computing Frontiers 2013
*Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability
Approach

- “Sample-and-model”
  - send a small portion of the available work to each device and run in parallel
  - Use the execution time of initial work to build a model to partition remaining work.
  - Partition at workgroup level

Results show improvement over non-heterogeneous execution. Note different behavior for different applications, as some applications are more predictable than others.
Summary dynamic partitioning

- **Scenario:**
  - one-time run of a (large/complex) application
  - Single-kernel / independent kernel[s]

- **Principle**
  - Chunk work in sample intervals
  - Tune granularity
  - Trade-off between on-line trial runs and “good” runs

- **Efficiency**
  - Balance between sampling granularity and application variability
Run-time based systems

StarPU and OmpSS
Role of runtime systems

- Portability
  - Abstraction
  - Drivers, plugins

- Control
  - Resource mapping
  - Scheduling

- Adaptiveness
  - Load balancing
  - Monitoring, sampling, calibrating

- Optimization
  - Requests aggregation
  - Resource locality
  - Computation offload
  - Computation/transfer overlap

Diagram showing HPC Applications, Parallel Compilers, Parallel Libraries, StarPU, and Drivers (CUDA, OpenCL) for CPU, GPU, and other devices.
Runtime systems for heterogeneous systems are task-based

- Distribute tasks at run-time between different *PU in the platform.

Thus, they must:

- Adapt to heterogeneity
  - Decide about tasks to offload
  - Decide about tasks to keep on CPU
- Take communication into account
  - Send computation requests
  - Send data to be processed
  - Fetch results back
- Decide about worthiness
  - Heterogeneous or not?
StarPU Programming Model: Sequential Task Submission
- The runtime will schedule them in parallel, when possible.

Dynamic DAG generation
- Tasks are submitted asynchronously at run-time
- Data references are annotated
- StarPU infers data dependences and builds a graph of tasks

StarPU Execution Model: Task Scheduling
- Map the graph of tasks (DAG) on the hardware
- Allocate computing resources
- Enforce dependency constraints
- Handle data transfers
StarPU specifics

- Specific extensions to C/C++ to express tasks, data collections, interaction with devices
- Performance models to decide runtime mapping
  - Learning-based
- Data transfer optimization
  - Prediction and prefetching
  - No unnecessary data transfers
- Heterogeneity: declare device-specific kernels
  - All registered in the run-time
  - RT Chooses device + selects device specific code
OmpSs

- Pragma-based programming model for heterogeneous system
- Input: sequential program + pragma’s
  - Annotations to determine data dependencies and tasks to be offloaded.
- Different kernels for different targets
- Uses: task-based run-time system
  - DAG created at runtime, to tackle dependencies
  - Mapping of DAG to heterogeneous platforms

Same principle as StarPU. Subtle differences with StarPU: programming model and back-end optimizations.
Heterogeneous Computing

- Single kernel
- Multi-kernel (complex) DAG

- Static
- Dynamic

- Qilin
- Insieme
- Glinda
- M2-Glinda*
- Totem (graph processing)
- SKMD
- GlassWing (MapReduce)

- Sporadic attempts
- Light runtime systems
- StarPU
- OmpSS
- ...

*M2-Glinda = multi-kernel, multi-GPU
Take home message [1]

- Heterogeneous computing works!
  - More resources.
  - Specialized resources.

- Most efforts in static partitioning and run-time systems
  - Glinda = static partitioning for single-kernel, data parallel applications
    - Now works for multi-kernel applications, too
  - StarPU, OmpSS = run-time based dynamic partitioning for multi-kernel, complex DAG applications

- Domain-specific efforts, too
  - Totem – graph processing
  - GlassWing – MapReduce
Choose a system based on your application scenario:

- Single-kernel vs. multi-kernel
- Massive parallel vs. Data-dependent
- Single run vs. Multiple run
- Programming model of choice

There are models to cover combinations of these choices!

- No framework to combine them all – food for thought?