Programming Models and Runtime Systems for Heterogeneous Architectures

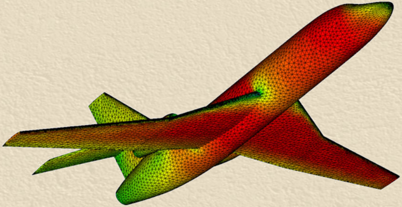
Sylvain Henry

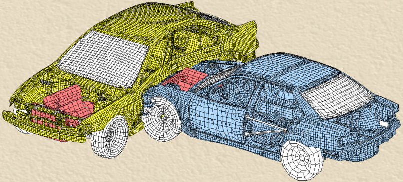
sylvain.henry@inria.fr

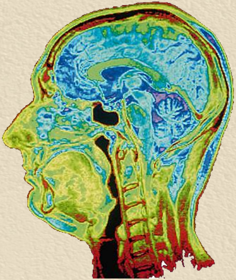
Advisors: Denis Barthou and Alexandre Denis

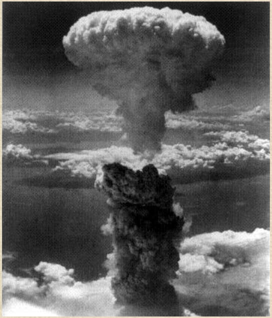
November 14, 2013

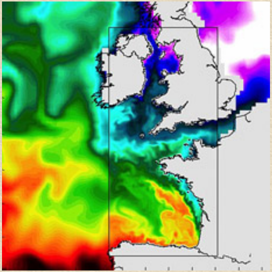
High-Performance Computing











Sources: Dassault aviation, BMW, Larousse, Interstices

Evolution of the architecture models

Parallel architectures

- Single-core architecture improvement stalled since 2003
  - Power wall: increasing the processor frequency exponentially increases power consumption
  - Memory wall: increasing gap between memory and processor speeds

Evolution of the architecture models

Parallel architectures

- Single-core architecture improvement stalled since 2003
  - Power wall: increasing the processor frequency exponentially increases power consumption
  - Memory wall: increasing gap between memory and processor speeds
- The number of transistors on a chip keeps increasing
  - Increase in the number of cores per chip
  - Multi-core architectures are omnipresent

Evolution of the architecture models

Parallel architectures

- Single-core architecture improvement stalled since 2003
  - Power wall: increasing the processor frequency exponentially increases power consumption
  - Memory wall: increasing gap between memory and processor speeds
- The number of transistors on a chip keeps increasing
  - Increase in the number of cores per chip
  - Multi-core architectures are omnipresent
- Trend
  - Multi-core with lower frequencies and more cores

Evolution of the architecture models

Specialized parallel architectures

- ➔ Cell Broadband Engine (2005)
  - ➔ 8 co-processors
  - ➔ Used in PlayStation 3 and in super-computers

Evolution of the architecture models

Specialized parallel architectures

- Cell Broadband Engine (2005)
  - 8 co-processors
  - Used in PlayStation 3 and in super-computers
- Graphics Processing Units (GPU)
  - Massively parallel architectures
  - Used to perform scientific computations

Evolution of the architecture models

Specialized parallel architectures

- Cell Broadband Engine (2005)
  - 8 co-processors
  - Used in PlayStation 3 and in super-computers
- Graphics Processing Units (GPU)
  - Massively parallel architectures
  - Used to perform scientific computations
- System-on-chip (SoC)
  - e.g. ARM, AMD Fusion
  - Integrated CPU, GPU, DSP...

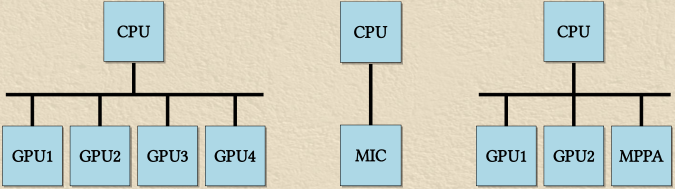
Evolution of the architecture models

Specialized parallel architectures

- Cell Broadband Engine (2005)
  - 8 co-processors
  - Used in PlayStation 3 and in super-computers
- Graphics Processing Units (GPU)
  - Massively parallel architectures
  - Used to perform scientific computations
- System-on-chip (SoC)
  - e.g. ARM, AMD Fusion
  - Integrated CPU, GPU, DSP...
- Trend: heterogeneous architectures
  - Composition of different architecture models

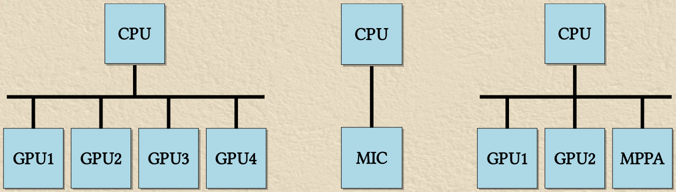
Heterogeneous architectures

- Multi-core (CPU) + several accelerators
- Most general case
  - Any number of accelerators
  - Any kind of accelerator
  - Any kind of interconnection network
- Examples:



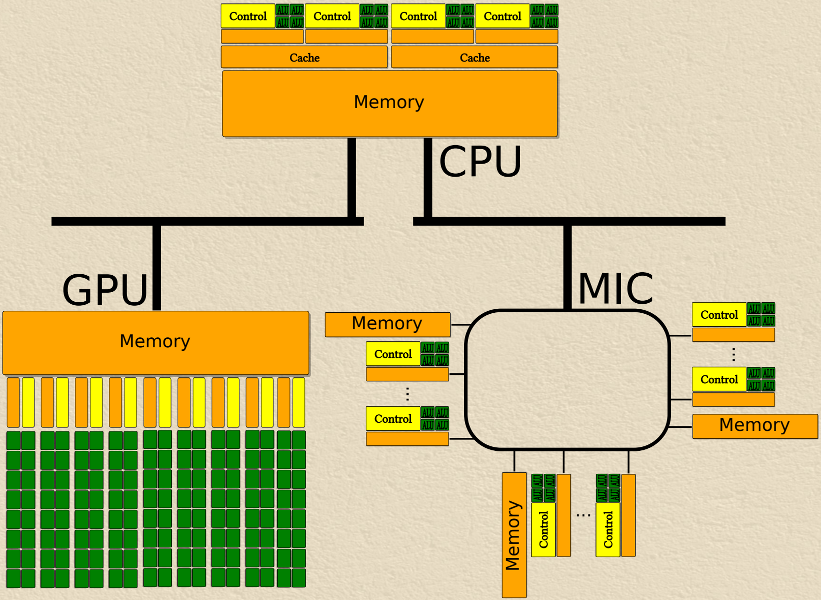
Heterogeneous architectures

- Multi-core (CPU) + several accelerators
- Most general case
  - Any number of accelerators
  - Any kind of accelerator
  - Any kind of interconnection network
- Examples:

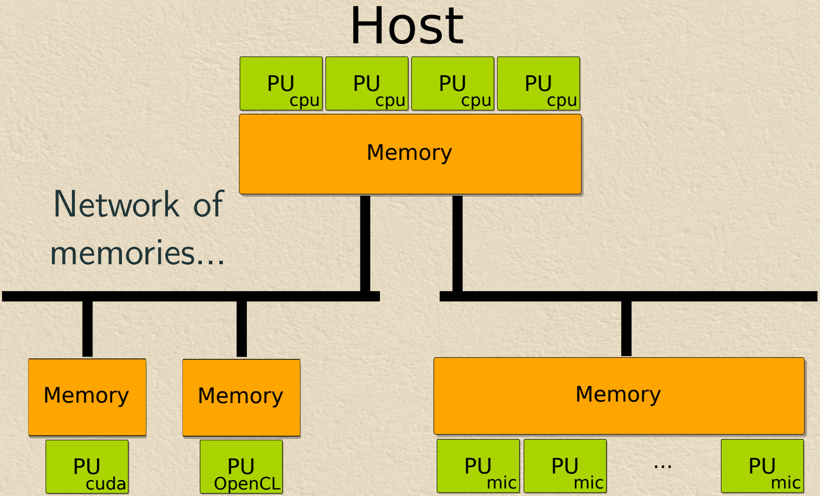


- Use best suited processing unit for each computation
- Manual tuning has to be repeated for each architecture
- Code portability difficult to achieve

Abstract architecture model



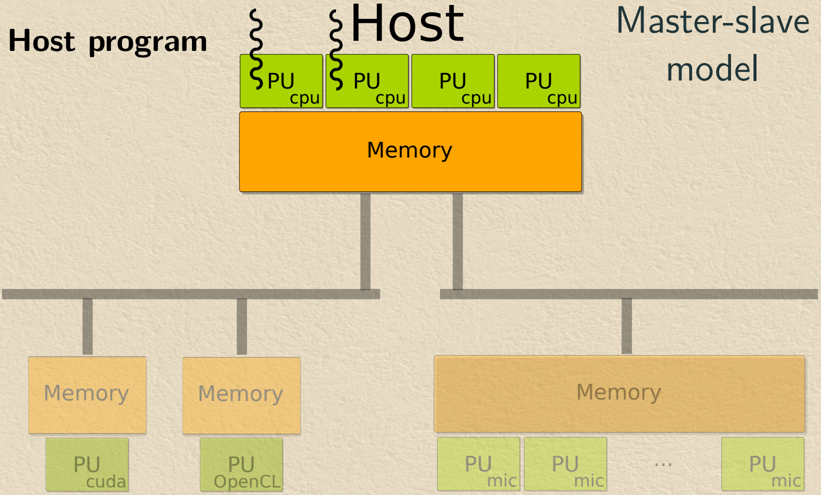
Abstract architecture model



...with associated heterogeneous processing units

Execution model

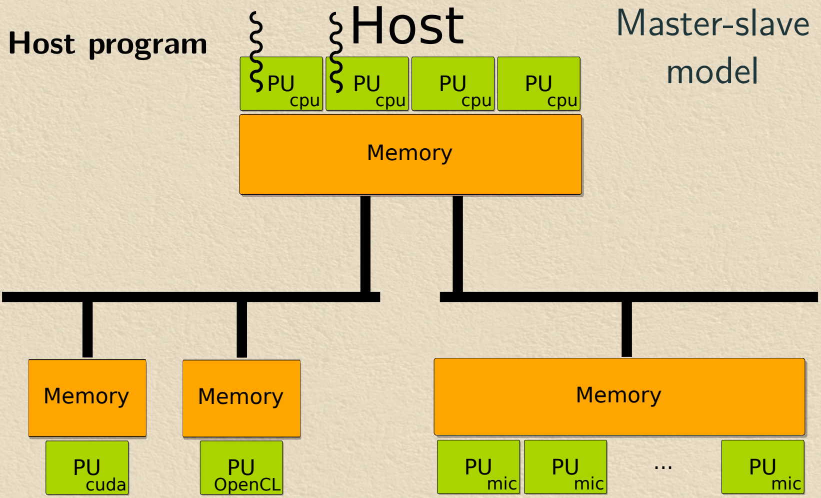
Host program



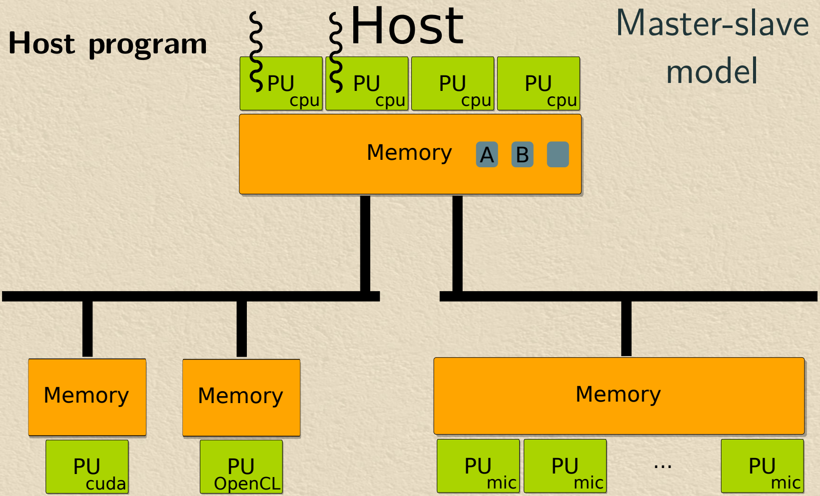
Master-slave model

Execution model

Master-slave model

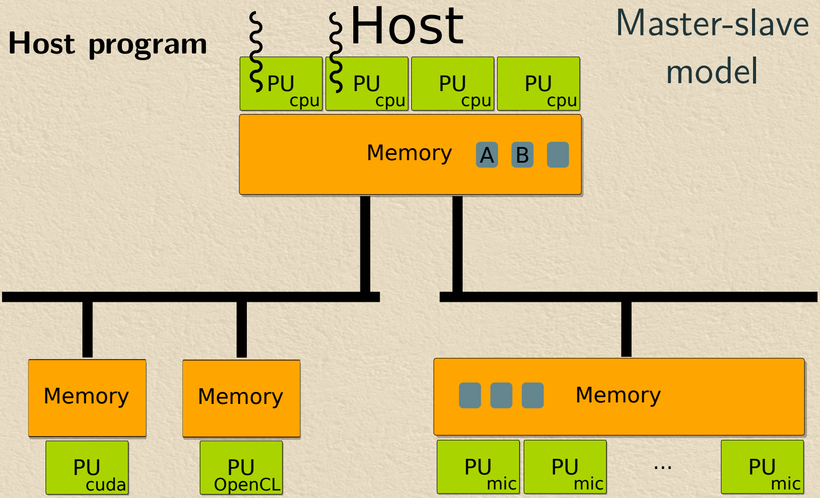


Execution model

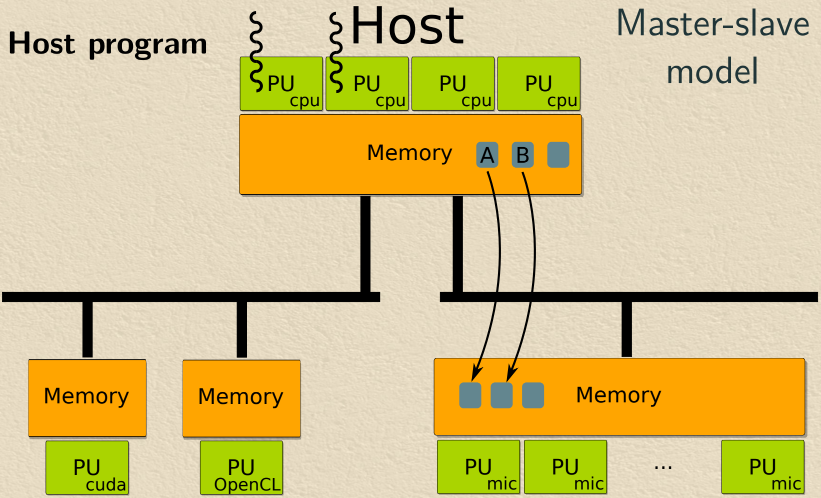


Master-slave model

Execution model

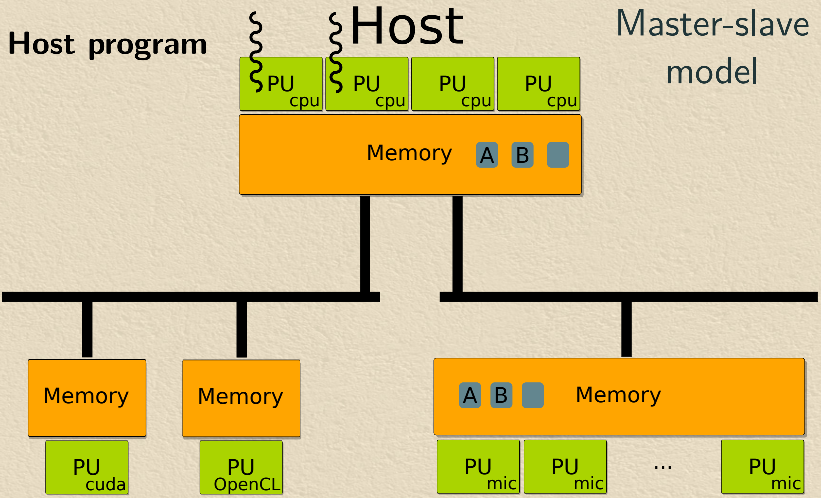


Execution model



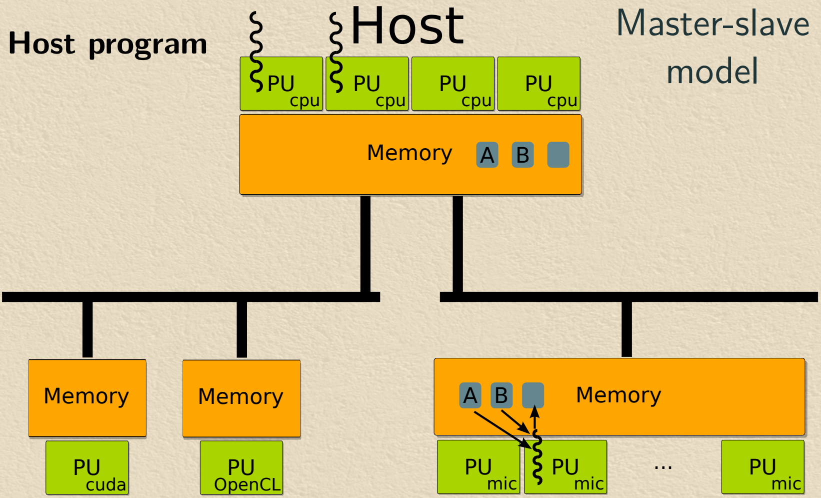
Master-slave model

Execution model



Master-slave model

Execution model

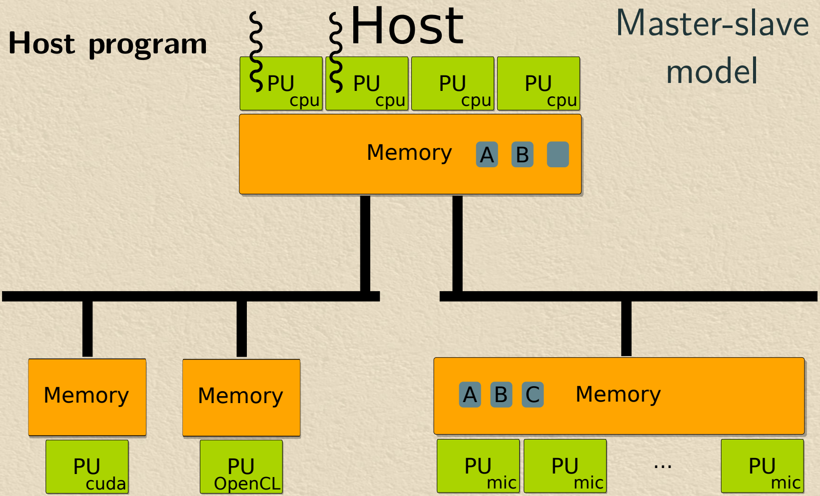


Master-slave model

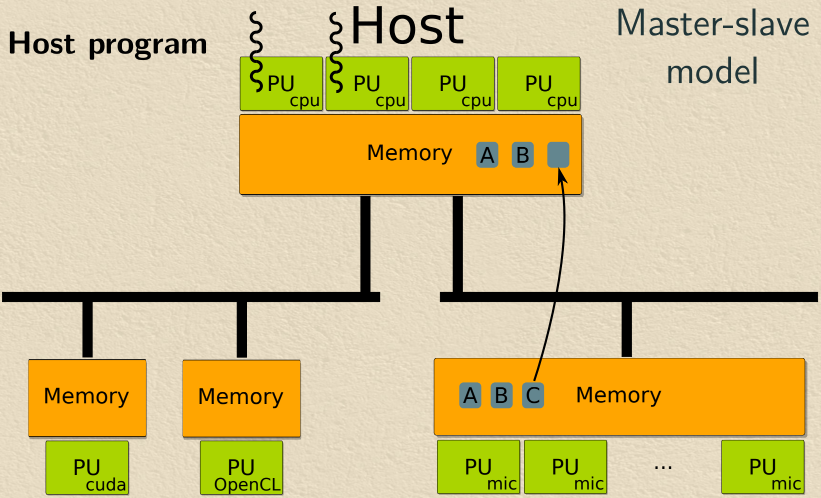
Execution model

Host program

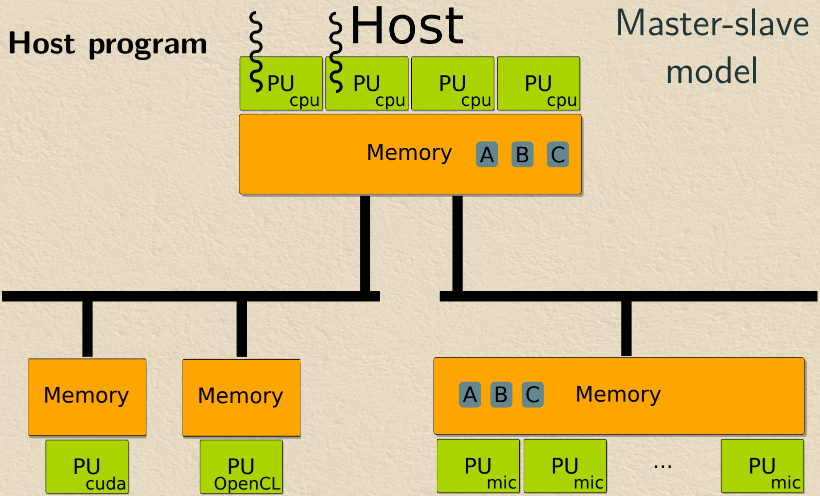
Master-slave model



Execution model

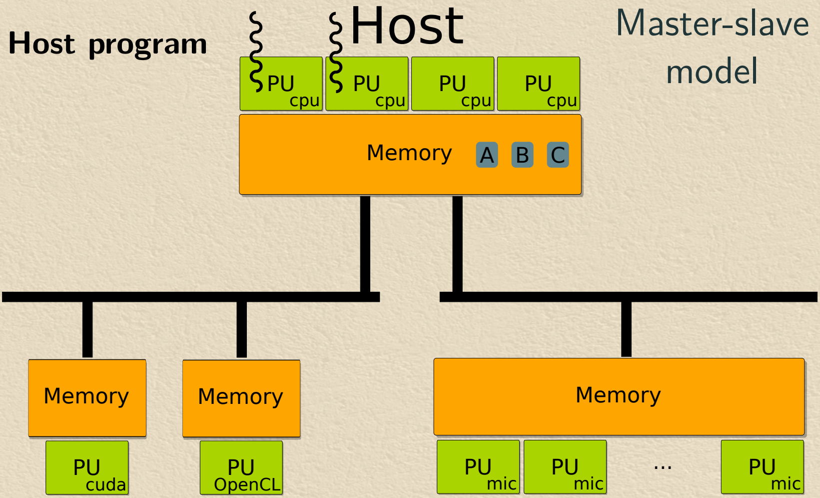


Execution model



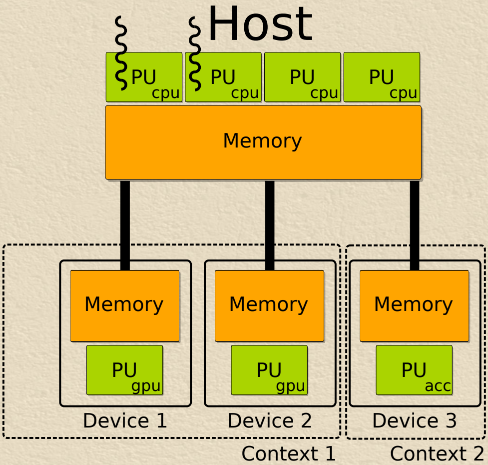
Master-slave model

Execution model



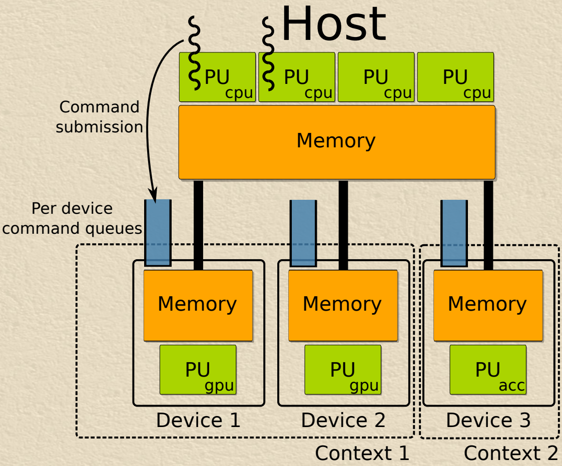
Programming model

Low-level approach (e.g. OpenCL, CUDA...)



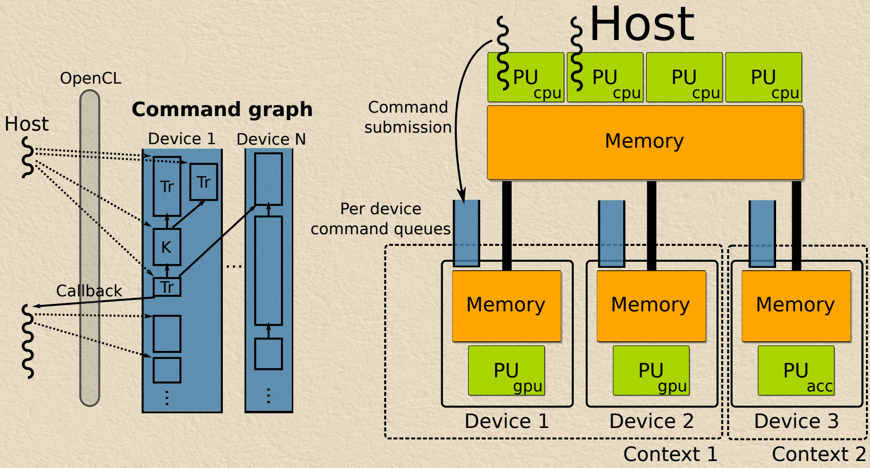
Programming model

Low-level approach (e.g. OpenCL, CUDA...)



Programming model

Low-level approach (e.g. OpenCL, CUDA...)



OpenCL example (uncluttered)

$C \leftarrow A + B$

```
float A[256], B[256], C[256];

clGetPlatformIDs(&platforms ...);
clGetDeviceIDs(platforms[0], &devices ...);
cl_context context = clCreateContext(devices ...);
cl_command_queue cq = clCreateCommandQueue(context, devices[0]...);

cl_mem bufA = clCreateBuffer(context, 1024...);
cl_mem bufB = clCreateBuffer(context, 1024...);
cl_mem bufC = clCreateBuffer(context, 1024...);

clEnqueueWriteBuffer(cq, bufA, 0, 1024, A, NULL, &event1...);
clEnqueueWriteBuffer(cq, bufB, 0, 1024, B, NULL, &event2...);

clSetKernelArg(kernelAdd, 0, sizeof(cl_mem), &bufA);
clSetKernelArg(kernelAdd, 1, sizeof(cl_mem), &bufB);
clSetKernelArg(kernelAdd, 2, sizeof(cl_mem), &bufC);
cl_event deps[] = {event1,event2};
clEnqueueNDRangeKernel(cq, kernelAdd, deps, &event3...);

clEnqueueReadBuffer(cq, bufC, 0, 1024, C, &event3, &event4);

clWaitForEvents(event4);

clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
```

Select accelerator

Allocate buffers

Send data

Execute kernel

Receive data

Release buffers

OpenCL simple multi-device example (NVIDIA)

```
const unsigned int MAX_GPU_COUNT = 8;
const unsigned int DATA_N = 1048576 * 24;
const unsigned int BLOCK_N = 128;
const unsigned int THREAD_N = 128;
const unsigned int ACCUM_N = BLOCK_N * THREAD_N;

int main(int argc, const char **argv)
{
    cl_context cxGPUContext;
    cl_device_id cdDevice;
    int deviceNr[MAX_GPU_COUNT];
    cl_command_queue commandQueue[MAX_GPU_COUNT];
    cl_mem d_Data[MAX_GPU_COUNT];
    cl_mem d_Result[MAX_GPU_COUNT];
    cl_program cpProgram;
    cl_kernel reduceKernel[MAX_GPU_COUNT];
    cl_event GPUDone[MAX_GPU_COUNT];
    cl_event GPUExecution[MAX_GPU_COUNT];
    cl_uint ciDeviceCount = 0;
    size_t programLength;
    cl_int ciErrNum;
    char cDeviceName [256];
    cl_mem h_DataBuffer;

    float h_SumGPU[MAX_GPU_COUNT * ACCUM_N];
    float *h_Data;
    double sumGPU;
    double sumCPU, dRelError;

    h_Data = (float *)malloc(DATA_N * sizeof(float));
    shrFillArray(h_Data, DATA_N);

    cxGPUContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL,
                                           NULL, &ciErrNum);
    if(shrCheckCmdLineFlag(argc, argv, "device"))
    {
        // User specified GPUs
        char* deviceList;
        char* deviceStr;
        char* next_token;

        // Create command queues for all Requested GPU's
        while(deviceStr != NULL)
        {
            // get & log device index # and name
            deviceNr[ciDeviceCount] = atoi(deviceStr);
            cdDevice = oclGetDev(cxGPUContext, deviceNr[ciDeviceCount]);
            ciErrNum = clGetDeviceInfo(cdDevice, CL_DEVICE_NAME,
            sizeof(cDeviceName), cDeviceName, NULL);
            shrCheckError(ciErrNum, CL_SUCCESS);

            // create a command que
            commandQueue[ciDeviceCount] = clCreateCommandQueue(cxGPUContext,
                                          cdDevice, 0, &ciErrNum);
            shrCheckError(ciErrNum, CL_SUCCESS);

            #ifdef GPU_PROFILING
                ciErrNum = clSetCommandQueueProperty(commandQueue[ciDeviceCount],
                                CL_QUEUE_PROFILING_ENABLE, CL_TRUE, NULL);
                shrCheckError(ciErrNum, CL_SUCCESS);
            #endif

            ++ciDeviceCount;

            deviceStr = strtok (NULL, " ,.-");
        }

        free(deviceList);
    }
    else
    {
        // Find out how many GPU's to compute on all available GPUs
        size_t nDeviceBytes;
        ciErrNum = clGetContextInfo(cxGPUContext, CL_CONTEXT_DEVICES, 0,
                                    NULL, &nDeviceBytes);
        shrCheckError(ciErrNum, CL_SUCCESS);
        ciDeviceCount = (cl_uint)nDeviceBytes/sizeof(cl_device_id);

        for(unsigned int i = 0; i < ciDeviceCount; ++i )
        {
            // get & log device index # and name
            deviceNr[i] = i;
            cdDevice = oclGetDev(cxGPUContext, i);
            ciErrNum = clGetDeviceInfo(cdDevice, CL_DEVICE_NAME,
                       sizeof(cDeviceName), cDeviceName, NULL);
            shrCheckError(ciErrNum, CL_SUCCESS);

            // create a command que
            commandQueue[i] = clCreateCommandQueue(cxGPUContext, cdDevice,
                              0, &ciErrNum);
            shrCheckError(ciErrNum, CL_SUCCESS);

            #ifdef GPU_PROFILING
                ciErrNum = clSetCommandQueueProperty(commandQueue[i],
                           CL_QUEUE_PROFILING_ENABLE, CL_TRUE, NULL);
                shrCheckError(ciErrNum, CL_SUCCESS);
            #endif
        }
    }

    // Load the OpenCL source code from the .cl file
    const char* source_path = shrFindFilePath("simpleMultiGPU.cl", argv[0]);
    char *source = oclLoadProgSource(source_path, "", &programLength);
    shrCheckError(source != NULL, shrTRUE);

    // Create the program for all GPUs in the context
    cpProgram = clCreateProgramWithSource(cxGPUContext, 1,
                (const char **)&source, &programLength, &ciErrNum);
    shrCheckError(ciErrNum, CL_SUCCESS);

    // build the program
    ciErrNum = clBuildProgram(cpProgram, 0, NULL, "-cl-mad-enable", NULL, NULL);
    if (ciErrNum != CL_SUCCESS)
    {
        // write out standard error, Build Log and PTX, then cleanup and exit
        oclLogBuildInfo(cpProgram, oclGetFirstDev(cxGPUContext));
        oclLogPtx(cpProgram, oclGetFirstDev(cxGPUContext), "oclSimpleMultiGPU.ptx");
        shrCheckError(ciErrNum, CL_SUCCESS);
    }

    // Create host buffer with page-locked memory
    h_DataBuffer = clCreateBuffer(cxGPUContext,
                   CL_MEM_COPY_HOST_PTR | CL_MEM_ALLOC_HOST_PTR,
                   DATA_N * sizeof(float), h_Data, &ciErrNum);
    shrCheckError(ciErrNum, CL_SUCCESS);

    // Create buffers for each GPU, with data divided evenly among GPU's
    int sizePerGPU = DATA_N / ciDeviceCount;
    int workOffset[MAX_GPU_COUNT];
    int workSize[MAX_GPU_COUNT];
    workOffset[0] = 0;
    for(unsigned int i = 0; i < ciDeviceCount; ++i )
    {
        workSize[i] = (i != (ciDeviceCount - 1)) ?
                      sizePerGPU : (DATA_N - workOffset[i]);

        // Input buffer
        d_Data[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY,
                    workSize[i] * sizeof(float), NULL, &ciErrNum);
        shrCheckError(ciErrNum, CL_SUCCESS);

        // Copy data from host to device
        ciErrNum = clEnqueueCopyBuffer(commandQueue[i], h_DataBuffer,
                   d_Data[i], workOffset[i] * sizeof(float),
                   0, workSize[i] * sizeof(float), 0, NULL, NULL);

        // Output buffer
        d_Result[i] = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,
                      ACCUM_N * sizeof(float), NULL, &ciErrNum);
        shrCheckError(ciErrNum, CL_SUCCESS);

        // Create kernel
        reduceKernel[i] = clCreateKernel(cpProgram, "reduce", &ciErrNum);
        shrCheckError(ciErrNum, CL_SUCCESS);

        // Set the args values and check for errors
        ciErrNum |= clSetKernelArg(reduceKernel[i], 0, sizeof(cl_mem), &d_Result[i]);
        ciErrNum |= clSetKernelArg(reduceKernel[i], 1, sizeof(cl_mem), &d_Data[i]);
        ciErrNum |= clSetKernelArg(reduceKernel[i], 2, sizeof(int), &workSize[i]);
        shrCheckError(ciErrNum, CL_SUCCESS);

        workOffset[i + 1] = workOffset[i] + workSize[i];
    }

    // Set # of work items in work group and total in 1 dimensional range
    size_t localWorkSize[] = {THREAD_N};
    size_t globalWorkSize[] = {ACCUM_N};

    // Start timer and launch reduction kernel on each GPU, with data split between them
    for(unsigned int i = 0; i < ciDeviceCount; i++)
    {
        ciErrNum = clEnqueueNDRangeKernel(commandQueue[i], reduceKernel[i],
                   1, 0, globalWorkSize, localWorkSize,
                   0, NULL, &GPUExecution[i]);
        shrCheckError(ciErrNum, CL_SUCCESS);
    }

    // Copy result from device to host for each device
    for(unsigned int i = 0; i < ciDeviceCount; i++)
    {
        ciErrNum = clEnqueueReadBuffer(commandQueue[i], d_Result[i], CL_FALSE,
                   0, ACCUM_N * sizeof(float),
                   h_SumGPU + i * ACCUM_N, 0, NULL, &GPUDone[i]);
        shrCheckError(ciErrNum, CL_SUCCESS);
    }

    // Synchronize with the GPUs and do accumulated error check
    clWaitForEvents(ciDeviceCount, GPUDone);

    // Aggregate results for multiple GPU's and stop/log processing time
    sumGPU = 0;
    for(unsigned int i = 0; i < ciDeviceCount * ACCUM_N; i++)
    {
        sumGPU += h_SumGPU[i];
    }

    // cleanup
    free(source);
    free(h_Data);
    for(unsigned int i = 0; i < ciDeviceCount; ++i )
    {
        clReleaseKernel(reduceKernel[i]);
        clReleaseCommandQueue(commandQueue[i]);
    }
    clReleaseProgram(cpProgram);
    clReleaseContext(cxGPUContext);
}
```

Issue tackled in this thesis

How to write efficient and portable applications for heterogeneous architectures?

1. How to express parallelism?
   - ➔ Task concept: same operation, several implementations (for each architecture)
2. How to schedule tasks on available units?
3. How to manage manage memories and data transfers?
4. How to adapt granularity of tasks to available units?

Low-level approaches

1. Dynamic construction of a graph of commands
2. Explicit task scheduling
3. Explicit memory management
4. Manual adaptation to the architecture
   - ➔ Static OpenCL kernel partitioning (Grewe et al., 2011)

Examples: OpenCL, CUDA...

Offloading approaches

Principle: use a simpler architecture model

➔ best suited for a CPU + single accelerator

1. Identify code regions to offload on the accelerator
2. Scheduling on the accelerator or fallback on the CPU
3. Data transfers automatically performed
4. No need for granularity adaptation

Example: OpenACC, OpenHMPP, OmpSS...

➔ Similar to OpenMP

➔ Easier to migrate legacy C or Fortran codes

Dynamic task graph approaches

1. Dynamic construction of a task graph
2. Automatic task scheduling
3. Automatic memory management
4. No granularity adaptation

Examples: StarPU, StarSS, XKaapi...

Static task graph approaches

1. Static description of a task graph
2. Automatic task scheduling
3. Automatic memory management
4. Static transformations on the graph

Examples: DaGUE, StreamIt (synchronous data-flow)...

Limits of the current approaches

Codes written using OpenCL

➔ Cannot be easily adapted to use more advanced runtime systems

Dynamic approaches lack overview of the task graph

➔ Control performed in host code

Static approaches have limited expressiveness

➔ No control (if, etc.) in the task graph

Outline

1. Context of the work
2. Extending OpenCL for a better portability
3. Heterogeneous parallel functional programming model

Extending OpenCL for better portability

Objectives

- ➔ Automatic kernel scheduling
- ➔ Automatic memory management and data transfers
- ➔ Automatic granularity adaptation

Extending OpenCL for better portability

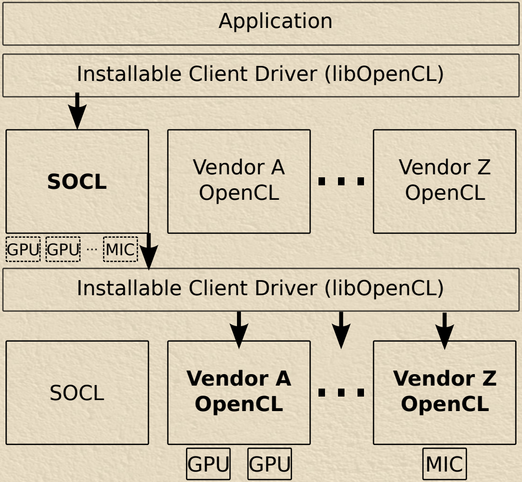
Objectives

- ➔ Automatic kernel scheduling
- ➔ Automatic memory management and data transfers
- ➔ Automatic granularity adaptation

SOCL: our extended OpenCL implementation

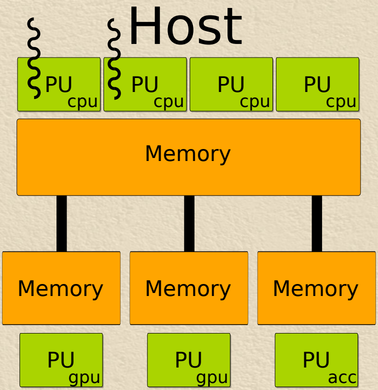
- ➔ Based on StarPU (**S**tarPU **O**pen**CL**)

SOCL unified platform overview

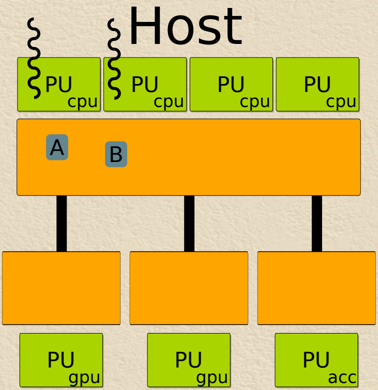


➔ Synchronizations between different platforms

SOCL: shared-object memory

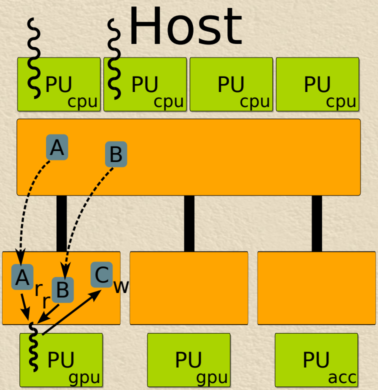


SOCL: shared-object memory



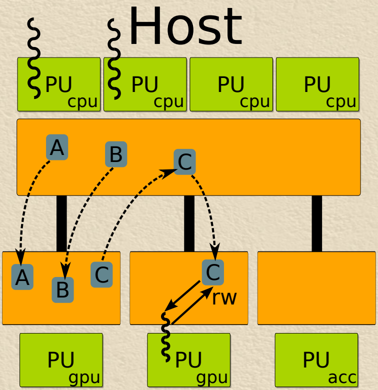
SOCL: shared-object memory

Automatic transfers



SOCL: shared-object memory

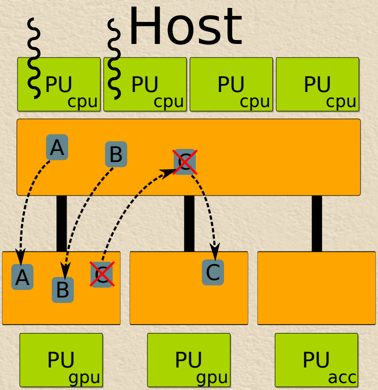
Automatic transfers



SOCL: shared-object memory

Automatic transfers

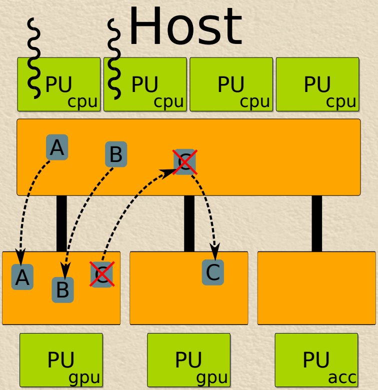
Coherence ensured



SOCL: shared-object memory

Automatic transfers

Coherence ensured



Relies on StarPU data management

SOCL: shared-object memory example

```
float A[256], B[256], C[256];

clGetPlatformIDs(&platforms ...);
clGetDeviceIDs(platforms [0], &devices ...);
cl_context context = clCreateContext(devices ...);

cl_command_queue cq1 = clCreateCommandQueue(context, devices[0]...);
cl_command_queue cq2 = clCreateCommandQueue(context, devices[1]...);

cl_mem bufA = clCreateBuffer(context, 1024...);
cl_mem bufB = clCreateBuffer(context, 1024...);
cl_mem bufC = clCreateBuffer(context, 1024...);
cl_mem bufC2 = clCreateBuffer(context, 1024...);

clEnqueueWriteBuffer(cq1, bufA, 0, 1024, A, NULL, &event1...);
clEnqueueWriteBuffer(cq1, bufB, 0, 1024, B, NULL, &event2...);

clSetKernelArg(kernelAdd, 0, sizeof(cl_mem), &bufA);
clSetKernelArg(kernelAdd, 1, sizeof(cl_mem), &bufB);
clSetKernelArg(kernelAdd, 2, sizeof(cl_mem), &bufC);
cl_event deps[] = {event1,event2};
clEnqueueNDRangeKernel(cq1, kernelAdd, deps, &event3...);

clEnqueueReadBuffer(cq1, bufC, 0, 1024, C, &event3, &event4);
clEnqueueWriteBuffer(cq2, bufC2, 0, 1024, C, &event3, &event5...);

clSetKernelArg( kernelPotrf , 0, sizeof(cl_mem), &bufC2);
clEnqueueNDRangeKernel(cq2, kernelPotrf, &event5, &event6...);

clWaitForEvents(event6);

clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
clReleaseMemObject(bufC2);
```

Select accelerators

Allocate buffers

Send data

Execute first kernel

Transfer data to GPU2

Execute second kernel

Release buffers

SOCL: shared-object memory example

```
float A[256], B[256], C[256];

clGetPlatformIDs(&platforms ...);
clGetDeviceIDs(platforms [0], &devices ...);
cl_context context = clCreateContext(devices ...);

cl_command_queue cq1 = clCreateCommandQueue(context, devices[0]...);
cl_command_queue cq2 = clCreateCommandQueue(context, devices[1]...);

cl_mem bufA = clCreateBuffer(context, 1024, CL_MEM_USE_HOST_PTR, A...);
cl_mem bufB = clCreateBuffer(context, 1024, CL_MEM_USE_HOST_PTR, B...);
cl_mem bufC = clCreateBuffer(context, 1024...);
```

Select accelerators

Allocate buffers

```
clSetKernelArg(kernelAdd, 0, sizeof (cl_mem), &bufA);
clSetKernelArg(kernelAdd, 1, sizeof (cl_mem), &bufB);
clSetKernelArg(kernelAdd, 2, sizeof (cl_mem), &bufC);

clEnqueueNDRangeKernel(cq1, kernelAdd, NULL, &event1 ...);
```

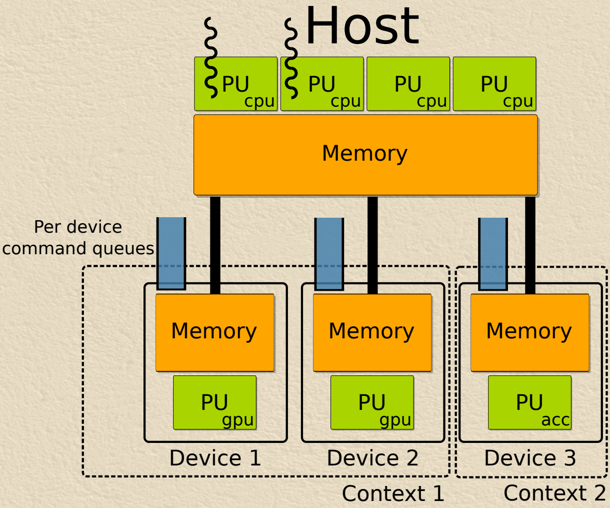
Execute first kernel

```
clSetKernelArg( kernelPotrf , 0, sizeof (cl_mem), &bufC);
clEnqueueNDRangeKernel(cq2, kernelPotrf, &event1 , &event2 ...);

clWaitForEvents( event2 );

clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
```

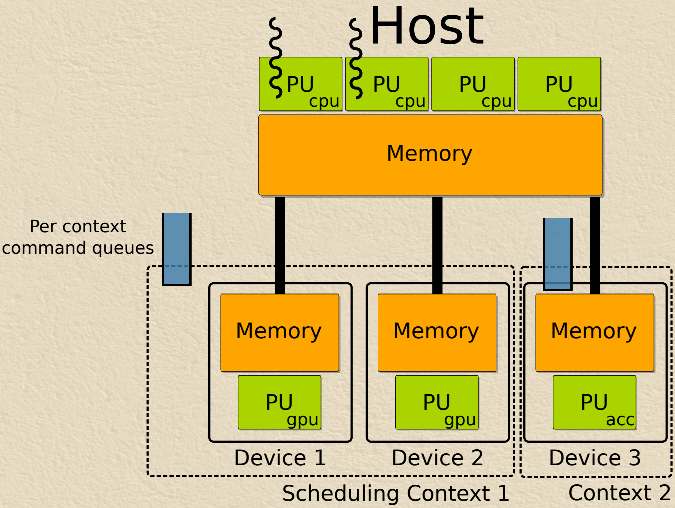
Execute second kernel

Release buffers

SOCL: context queues

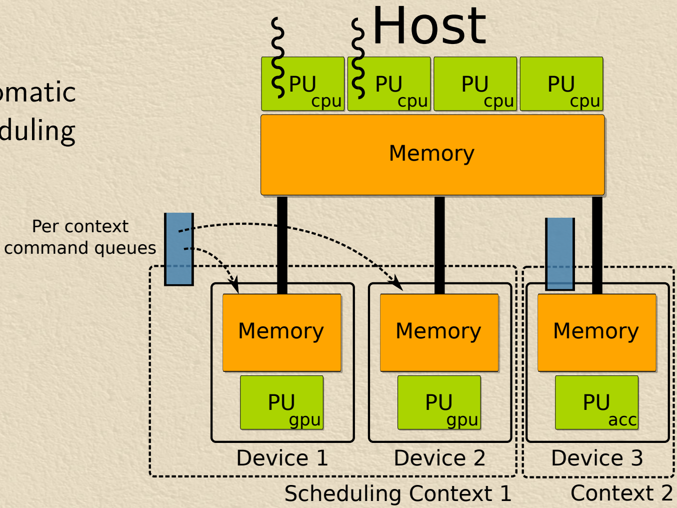


SOCL: context queues



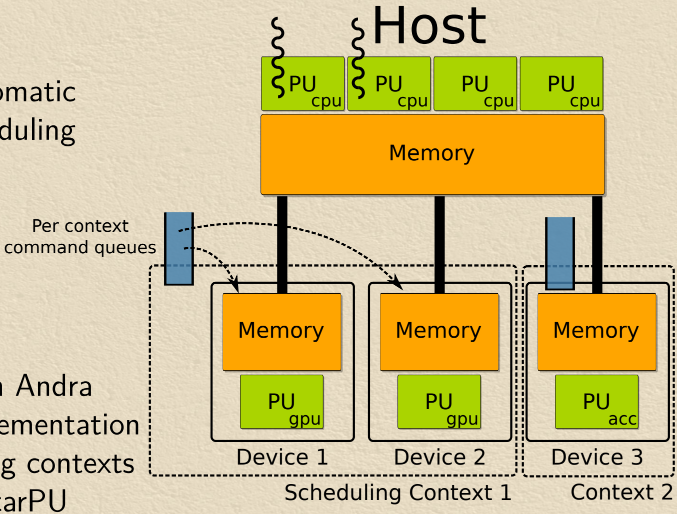
SOCL: context queues

Automatic scheduling



SOCL: context queues

Automatic scheduling



Relies on Andra Hugo's implementation of scheduling contexts into StarPU

SOCL: context queues example

```
float A[256], B[256], C[256];

clGetPlatformIDs(&platforms ...);
clGetDeviceIDs(platforms [0], &devices ...);
cl_context context = clCreateContext(devices ...);

cl_command_queue cq1 = clCreateCommandQueue(context, devices[0]...);
cl_command_queue cq2 = clCreateCommandQueue(context, devices[1]...);

cl_mem bufA = clCreateBuffer(context, 1024, CL_MEM_USE_HOST_PTR, A...);
cl_mem bufB = clCreateBuffer(context, 1024, CL_MEM_USE_HOST_PTR, B...);
cl_mem bufC = clCreateBuffer(context, 1024...);
```

Select accelerators

Allocate buffers

```
clSetKernelArg(kernelAdd, 0, sizeof (cl_mem), &bufA);
clSetKernelArg(kernelAdd, 1, sizeof (cl_mem), &bufB);
clSetKernelArg(kernelAdd, 2, sizeof (cl_mem), &bufC);

clEnqueueNDRangeKernel(cq1, kernelAdd, NULL, &event1...);
```

Execute first kernel

```
clSetKernelArg( kernelPotrf , 0, sizeof (cl_mem), &bufC);
clEnqueueNDRangeKernel(cq2, kernelPotrf, &event1, &event2...);

clWaitForEvents(event2);

clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
```

Execute second kernel

Release buffers

SOCL: context queues example

```
float A[256], B[256], C[256];

clGetPlatformIDs(&platforms ...);
clGetDeviceIDs(platforms[0], &devices ...);
cl_context context = clCreateContext(devices ...);

cl_command_queue cq = clCreateCommandQueue(context, NULL...);
```

Select accelerators

```
cl_mem bufA = clCreateBuffer(context, 1024, CL_MEM_USE_HOST_PTR, A...);
cl_mem bufB = clCreateBuffer(context, 1024, CL_MEM_USE_HOST_PTR, B...);
cl_mem bufC = clCreateBuffer(context, 1024...);
```

Allocate buffers

```
clSetKernelArg(kernelAdd, 0, sizeof(cl_mem), &bufA);
clSetKernelArg(kernelAdd, 1, sizeof(cl_mem), &bufB);
clSetKernelArg(kernelAdd, 2, sizeof(cl_mem), &bufC);

clEnqueueNDRangeKernel(cq, kernelAdd, NULL, &event1...);
```

Execute first kernel

```
clSetKernelArg(kernelPotrf, 0, sizeof(cl_mem), &bufC);
clEnqueueNDRangeKernel(cq, kernelPotrf, &event1, &event2...);

clWaitForEvents(event2);
```

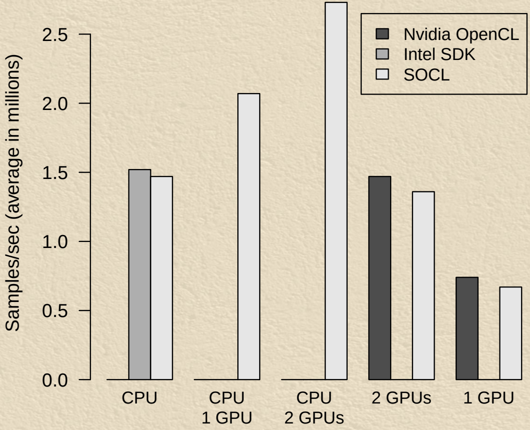
Execute second kernel

```
clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
```

Release buffers

SOCL: some benchmarks

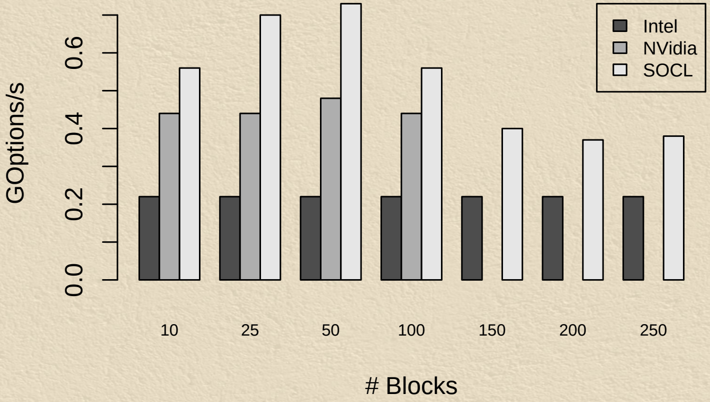
LuxRender (rendering software)



Averell1: Intel Xeon E5-2650 2.00GHz with 64GB, 2 NVidia Tesla M2075

SOCL: some benchmarks

Black Scholes - blocks of 5M options



Hannibal: Intel Xeon X5550 2.67GHz with 24GB, 3 NVidia Quadro FX 5800

➔ automatic handling of large problem sizes

SOCL: granularity adaptation mechanism

Partitioning function (per kernel)

- ➔ Let users associate a partitioning function to kernels

Partitioning factors

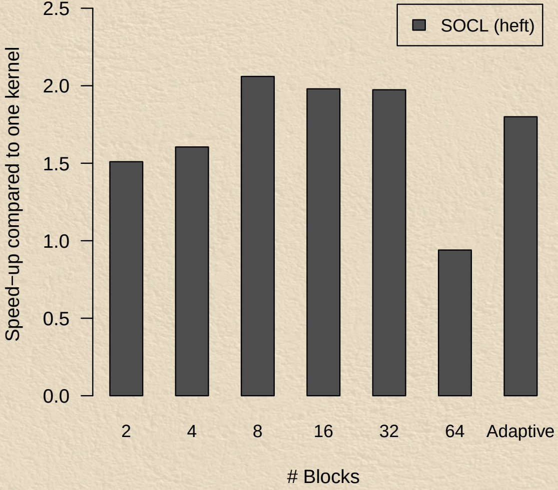
- ➔ Partitioning functions takes a partitioning factor as parameter
- ➔ Partitioning factor provided by the runtime system

Strategy

- ➔ Sample with different factors (in a given range)
- ➔ Select the best one

SOCL: some benchmarks

NBody (OTOO) - 20 iterations - 4000k particles



Hannibal: Intel Xeon X5550 2.67GHz with 24GB, 3 NVidia Quadro FX 5800

SOCL: implementation

- Full OpenCL 1.0 specification implementation
  - Some additional 1.1 and 1.2 APIs
  - Installable Client Driver (ICD) extension supported
- Integrated into StarPU's repository
  - `http://runtime.bordeaux.inria.fr/StarPU/`

SOCL: conclusion

1. OpenCL interface
2. Automatic task scheduling
   ➔ Command queues associated to contexts
3. Automatic memory management
4. Granularity adaptation mechanism
   ➔ Partitioning functions

Performance on par with state of the art

Publications

1. *Programmation multi-accélérateurs unifiée en OpenCL* - RenPAR'20 (2011)
2. *Programmation multi-accélérateurs unifiée en OpenCL* (extended) - TSI 31 (2012)
3. *SOCL: An OpenCL Implementation with Automatic Multi-Device Adaptation Support* - Inria Research Report (2013)

Outline

1. Context of the work
2. Extending OpenCL for a better portability
3. Heterogeneous parallel functional programming model

Heterogeneous parallel functional model

Objective

- Use a more declarative language to describe task graphs
  - Integrate control (if, loops, data-dependence…)
  - Allow static and dynamic transformations
  - Better granularity adaptation support

Heterogeneous parallel functional model

Objective

- Use a more declarative language to describe task graphs
  - Integrate control (if, loops, data-dependence...)
  - Allow static and dynamic transformations
  - Better granularity adaptation support

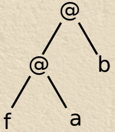
Use implicit parallel functional programming

- Kernels $\simeq$ pure functions
- Functional programs are graphs of pure functions

Functional programming

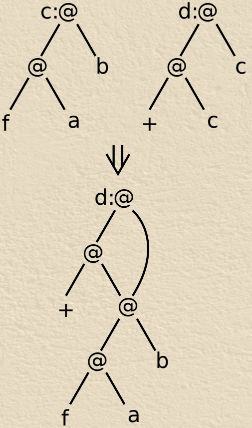
Application:

$$f \quad a \quad b$$



Constant applicative forms:

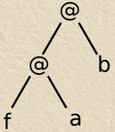
$$\begin{aligned} d &= c + c \\ c &= f \;\; a \;\; b \end{aligned}$$



Functional programming

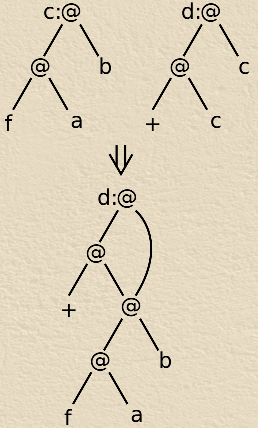
Application:

f a b



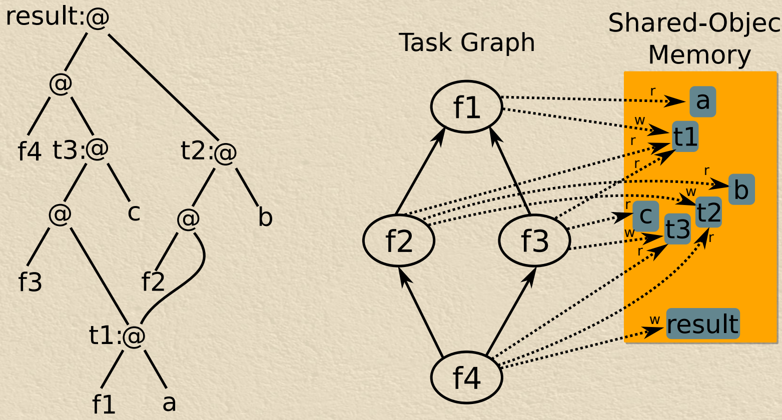
Constant applicative forms:

d = c + c
c = f a b



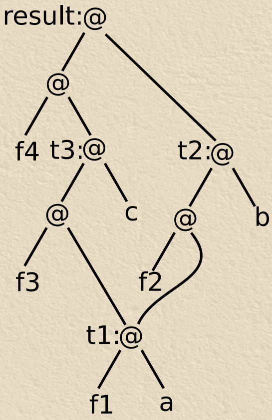
We can associate kernels to some symbols (e.g. "+", "f"): **data-flow graph**

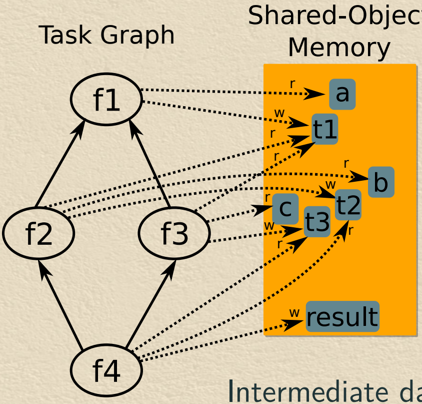
Parallel evaluation




```
result = f4 (f3 t1 c) (f2 t1 b)
t1 = f1 a
```

Parallel evaluation

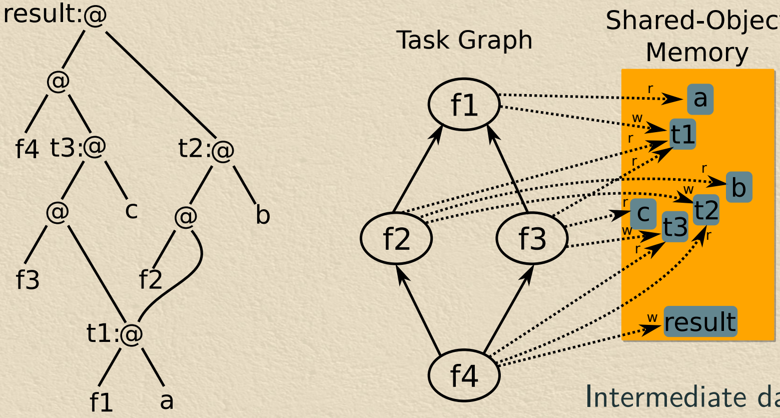




Intermediate data automatically allocated

```
result = f4 (f3 t1 c) (f2 t1 b)
t1 = f1 a
```

Parallel evaluation




```
result = f4 (f3 t1 c) (f2 t1 b)
t1 = f1 a
```

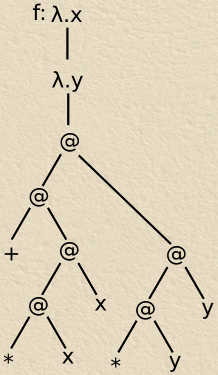
Intermediate data automatically allocated

Garbage collection of unused data

Control

Abstractions (functions)

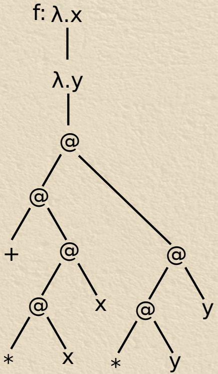
```
f x y = (x * x) + (y * y)
```



Control

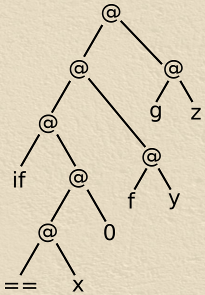
Abstractions (functions)

f x y = (x ∗ x) + (y ∗ y)



Conditionals

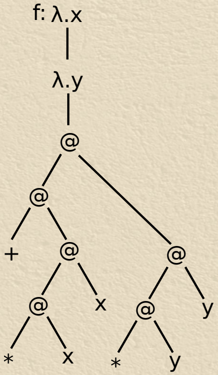
if x == 0 **then** f y **else** g z



Control

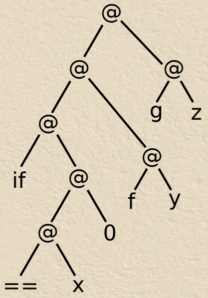
Abstractions (functions)

f x y = (x ∗ x) + (y ∗ y)



Conditionals

if x == 0 **then** f y **else** g z



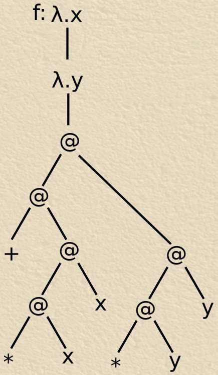
Recursive functions ≃ loops

while test f x = **if** test x **then** (while test f (f x)) **else** x

Control

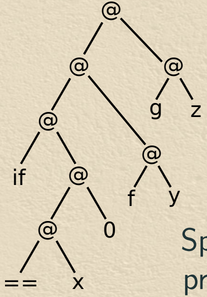
Abstractions (functions)

f x y = (x ∗ x) + (y ∗ y)



Conditionals

if x == 0 **then** f y **else** g z



Speculative prefetching

Speculative execution

Recursive functions ≃ loops

while test f x = **if** test x **then** (while test f (f x)) **else** x

Data-partitioning

split w h m

- Split matrix m in $w \times h$ tiles
- Result is a matrix of matrices

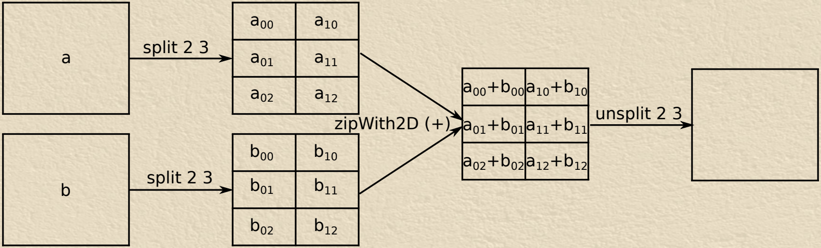
unsplit w h m

- Recompose matrix m
- m must be a $w \times h$ matrix of matrices
- Costly operation
  - Transfer all matrix parts in the same memory

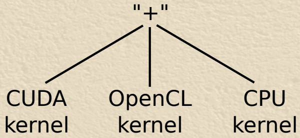
Data-partitioning example

Tiled matrix addition

```
addTiled a b = unsplit w h (zipWith2D (+)
                   (split w h a)
                   (split w h b))
```



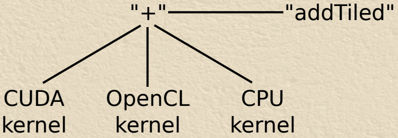
Granularity adaptation




```
addTiled a b =
    unsplit w h (zipWith2D (+)
        (split w h a)
        (split w h b))
```

➔ Cost models to select between kernels (cf StarPU, etc.)

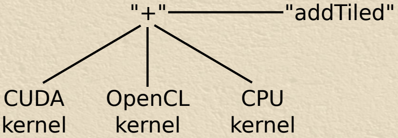
Granularity adaptation




```
addTiled a b =
    unsplit w h (zipWith2D (+)
        (split w h a)
        (split w h b))
```

- Cost models to select between kernels (cf StarPU, etc.)
- Can we extend them to select between kernels and alternative expression(s)?

Granularity adaptation




```
addTiled a b =
   unsplit w h (zipWith2D (+)
      (split w h a)
      (split w h b))
```

- ➔ Cost models to select between kernels (cf StarPU, etc.)
- ➔ Can we extend them to select between kernels and alternative expression(s)?
- ➔ Implemented strategy based on input data size

Transformations

Rewrite rules

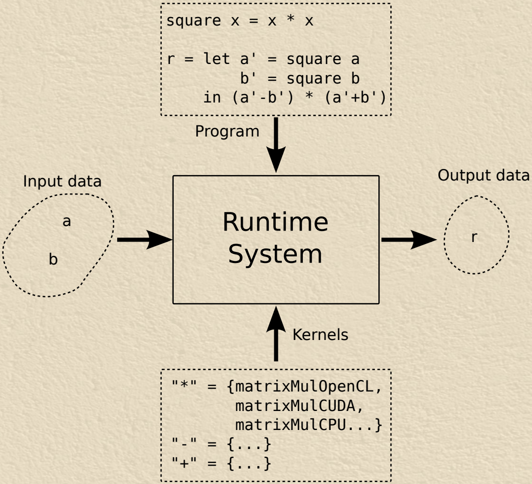
- Detect and modify patterns in the program/graph

Example: remove unnecessary data partitions

- forall w h . split w h (unsplit w h x) = x

```
r = a + b + c

r = (unsplit w h (zipWith2D (+) (split w h a) ( split w h b))) + c

r = unsplit w h (zipWith2D (+)
      ( split w h ( unsplit w h (zipWith2D (+) (split w h a) ( split w h b))))
      ( split w h c))

r = unsplit w h (zipWith2D (+)
      (zipWith2D (+) (split w h a) ( split w h b)))
      ( split w h c))
```

ViperVM: runtime system overview



Configuration

```
pf <- initPlatform $ Configuration {
        libraryOpenCL = "libOpenCL.so"
      }

rt <- initRuntime pf eagerScheduler

a <- initFloatMatrix rt [[1.0, 2.0, 3.0],
                         [4.0, 5.0, 6.0],
                         [7.0, 8.0, 9.0]]

b <- initFloatMatrix rt [[1.0, 4.0, 7.0],
                         [2.0, 5.0, 8.0],
                         [3.0, 6.0, 9.0]]

builtins <- loadBuiltins rt [
   ("+", floatMatrixAddBuiltin),
   ("-", floatMatrixSubBuiltin),
   ("*", floatMatrixMulBuiltin),
   ("a", dataBuiltin a),
   ("b", dataBuiltin b)]

prog <- readFile "example.vvm"
r <- eval builtins prog

printFloatMatrix rt r
```

3 kinds of codes

Configuration	Coordination	Computation
Host code (mostly imperative)	Parallel functional code	Kernels (C, Fortran, CUDA, OpenCL...)

Coordination

```
-- File: example.vvm

square x = x * x

main = let a' = square a
           b' = square b
       in (a'-b')*(a'+b')
```

Computation

```
__kernel void floatMatrixAdd(uint width, uint height,
   __global float* A, __global float* B, __global float* C){

  int gx = get_global_id(0);
  int gy = get_global_id(1);

  if (gx < width && gy < height) {
    C[gy*width+gx] = A[gy*width+gx] + B[gy*width+gx];
  }
}
```

ViperVM: expressivity

Tiled matrix addition example

```
/* StarPU */
struct starpu_data_filter f = {
   .filter_func = starpu_matrix_filter_vertical_block,
   .nchildren = w
};

struct starpu_data_filter f2 = {
   .filter_func = starpu_matrix_filter_block,
   .nchildren = h
};

starpu_data_map_filters(a, 2, &f, &f2);
starpu_data_map_filters(b, 2, &f, &f2);
starpu_data_map_filters(c, 2, &f, &f2);

for (i=0; i<nw; i++) { for (j=0; j<nh; j++) {
   starpu_data_handle_t sa = starpu_data_get_sub_data(a, 2, i, j);
   starpu_data_handle_t sb = starpu_data_get_sub_data(b, 2, i, j);
   starpu_data_handle_t sc = starpu_data_get_sub_data(c, 2, i, j);

   starpu_insert_task(&add, STARPU_R, sa, STARPU_R, sb, STARPU_W, sc, 0);
}}

starpu_task_wait_for_all();
starpu_data_unpartition(c,0);
```

```
-- ViperVM: explicit
c = unsplit (zipWith2D (+)
      (split w h a)
      (split w h b))
```

```
-- ViperVM: with automatic granularity adaptation
c = a + b
```

ViperVM: some (preliminary) benchmarks

Matrix addition (tile size = 8k)

Dimensions	ViperVM 3 GPUs+CPU	ViperVM 3 GPUs	StarPU 3 GPUs
16K x 16K	1.9s	2.1s	1.4s
24K x 24K	4.0s	4.4s	2.9s

Matrix multiplication (4096x4096)

w x h	1024x1024	4096x1024	1024x4096
GPU (1x)	4.5s	4.4s	4.3s
GPU (2x)	3.6s	2.9s	3.2s
GPU (3x)	3.1s	2.5s	3.3s
CPU	31s	36s	35s
GPU (3x) + CPU	3.3s	3.7s	10s

- ➔ Performance comparable with StarPU
- ➔ Scales with the number of devices
  - ➔ Scheduling policy not on par with StarPU's ones

ViperVM implementation

- Alpha version 0.2
  - `https://github.com/hsyl20/HViperVM/tree/0.2`
- Runtime system implemented in Haskell
  - Lisp-like frontend (parser)
  - Parallel reducer (using Software Transactional Memory)
  - Support for OpenCL kernels
  - Eager scheduling strategy
  - Naive substitution mechanism (based on input sizes)
- Future works
  - Garbage collector
  - Other backends (CUDA, Xeon Phi. . .)
  - Better scheduling strategies (HEFT. . .)
  - Enhanced frontend (type checking, etc.)

Hetereogeneous parallel functional model

Conclusion

Parallel function programming + kernels

- ➔ Adapted language to describe task graphs
- ➔ Control integrated in the graph
- ➔ Native kernel performance
- ➔ Static and dynamic graph transformations
- ➔ Granularity adaptation mechanism

Publications

1. *ViperVM: a Runtime System for Parallel Functional High-Performance Computing on Heterogeneous Architectures* - FHPC workshop (2013)

General conclusion

Problem tackled

- ➔ Writing efficient and portable codes for heterogeneous architectures

Contributions

- ➔ Better portability for OpenCL applications with SOCL
  - ➔ Automatic memory management and kernel scheduling
- ➔ High-level approach using functional programming
  - ➔ Better expressivity
  - ➔ Graph transformations
- ➔ Granularity adaptation mechanisms in both cases

Perspectives

Improve granularity adaptation

- ➔ Cost models for functional expressions
- ➔ Inference of the partitionning factors
- ➔ Choose between several alternative expressions

Revisit common HPC issues in the heterogeneous parallel functional model

- ➔ Check-pointing
- ➔ Fault-tolerance

Kernel generation and transformation

- ➔ Data-parallel kernel description
- ➔ Automatic derivation of alternative algorithms
  - ➔ cf Bird-Meertens formalism