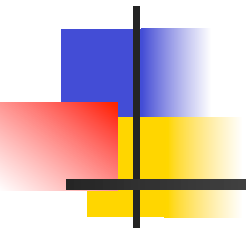
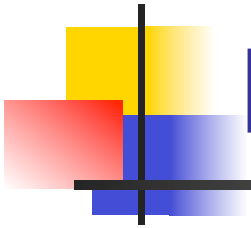
Towards a High Level Approach for the Programming of Heterogeneous Clusters

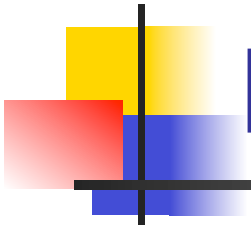


M. Viñas, B. B. Fraguela, D. Andrade, R. Doallo
Computer Architecture Group
Universidade da Coruña
Spain



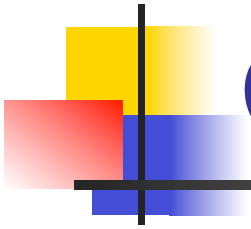
Motivation

- Heterogeneous clusters are being increasingly adopted
  - Large performance and power benefits
- They involved more programming effort
  - Distributed memory, both between nodes and host/devices
  - Accelerators are harder to program than traditional CPUs



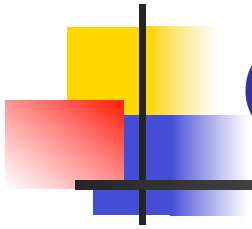
Motivation (II)

- Many proposals to tackle the programming of these systems. Often
  - Still many low level details exposed
  - SPMD processes
  - Task-parallelism
- Higher level approaches should be explored



Contribution

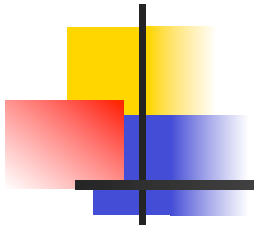
- Explore data-parallelism at cluster level combined with a tool for heterogeneous programming
- Achieved using
  - Hierarchically Tiled Array data type: arrays distributed by tiles on the cluster
  - Heterogeneous Programming Library: simple development of heterogeneous applications



Outline

- Hierarchically Tiled Arrays
- Heterogeneous Programming Library
- Integration
- Evaluation
- Conclusions and future work

Hierarchically Tiled Arrays (HTAs)

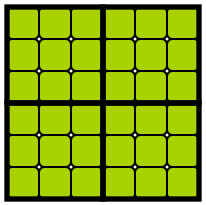


- Data type in a sequential language
- HTAs are tiled arrays where each tile can be a standard array or a tiled array. They provide
  - Data distribution in distributed-memory
    - Provides a global view
  - Locality
  - Data parallelism in operations on tiles
    - Single thread of control except in the data parallel operations
- Implementation based on C++ and MPI
  - http://polaris.cs.uiuc.edu/hta/

HTA Indexing

- Can choose ranges of tiles, scalars or combinations of both

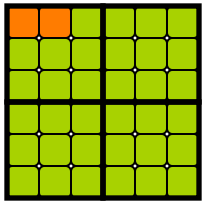
h



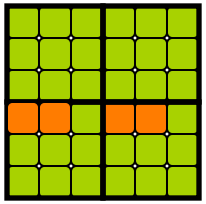
HTA<float, 2>::alloc({3, 3}, {2, 2});

h({0, 0:1})

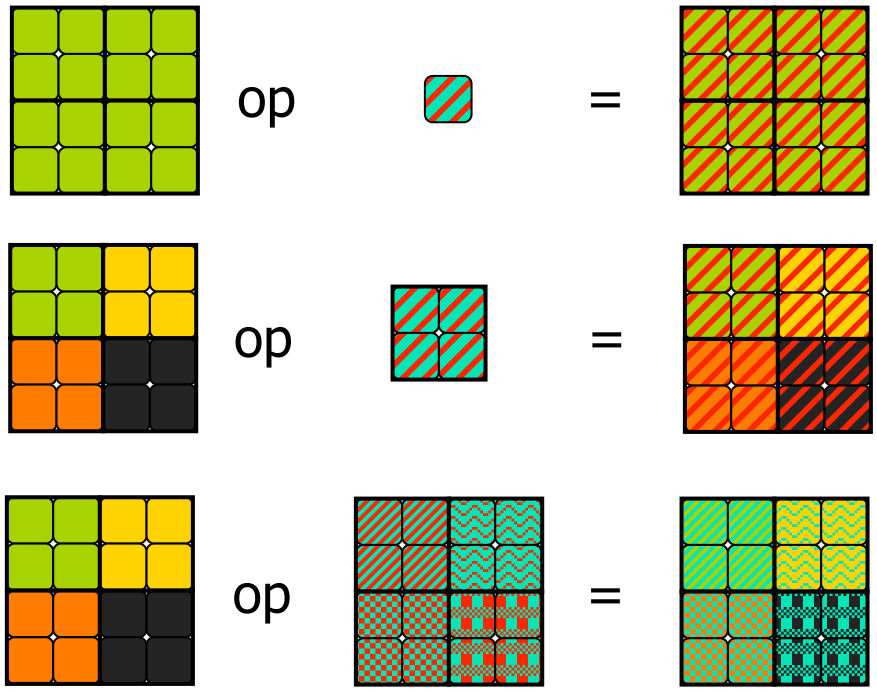
h[{0, 0:1}]

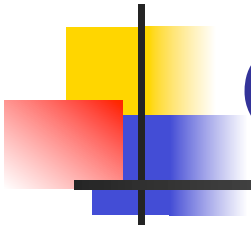


h({1, 0:1})[{0, 0:1}]



Operations on HTAs





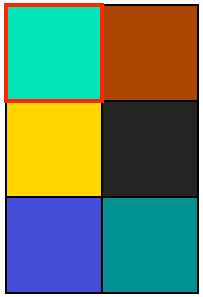
Operations on HTAs (II)

- Element-by-element arithmetic operations
- Operations typical of array languages
  - Matrix multiply, transposition, etc.
- Extended operator framework (reduce, mapReduce, scan) including user-defined operations
- Communications happen in assignments and provided array operations

Example

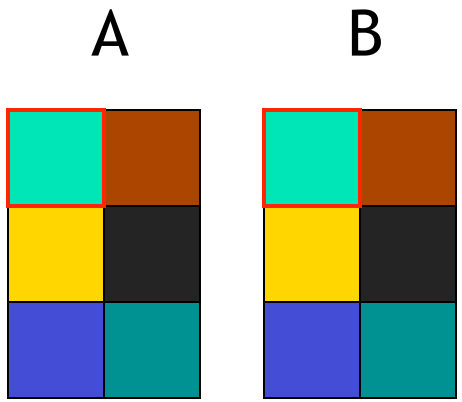
```
HTA<int, 2> A = HTA<int, 2>::alloc({N, N}, {3, 2});
```

A



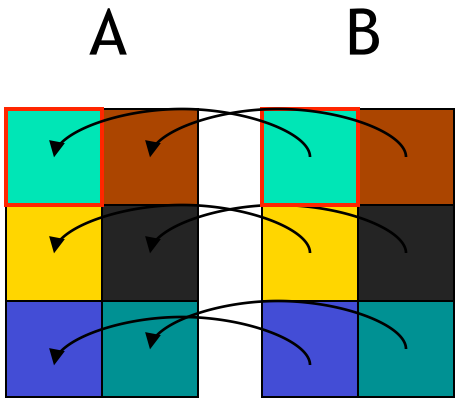
Example

```
HTA<int, 2> A = HTA<int, 2>::alloc({N, N}, {3, 2});
HTA<int, 2> B = A.clone();
```



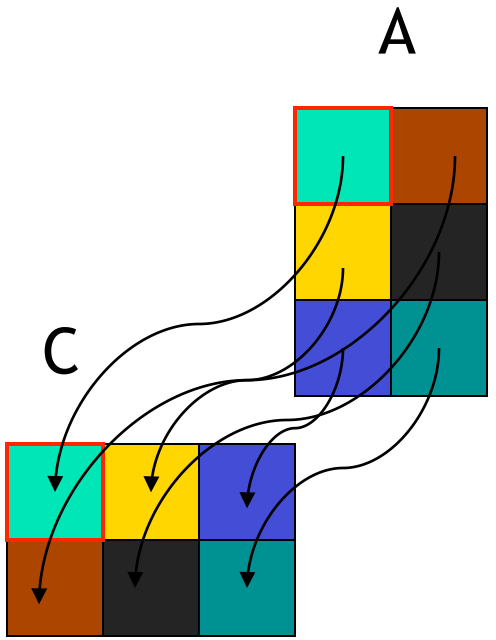
Example

```
struct Example {
  void operator() (HTA<int, 2> A, HTA<int, 2> B) {
    A = A + alpha * B;
  }
};

HTA<int, 2> A = HTA<int, 2>::alloc({N, N}, {3, 2});
HTA<int, 2> B = A.clone();
...
A = A + alpha * B;
hmap(Example(), A, B); //Same
```

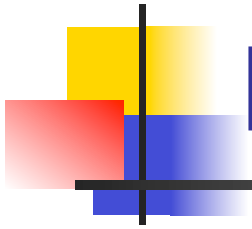


Example

```
struct Example {
  void operator() (HTA<int, 2> A, HTA<int, 2> B) {
    A = A + alpha * B;
  }
};

HTA<int, 2> A = HTA<int, 2>::alloc({N, N}, {3, 2});
HTA<int, 2> B = A.clone();
...
A = A + alpha * B;
hmap(Example(), A, B); //Same
...
HTA<int, 2> C = A.transpose();
```



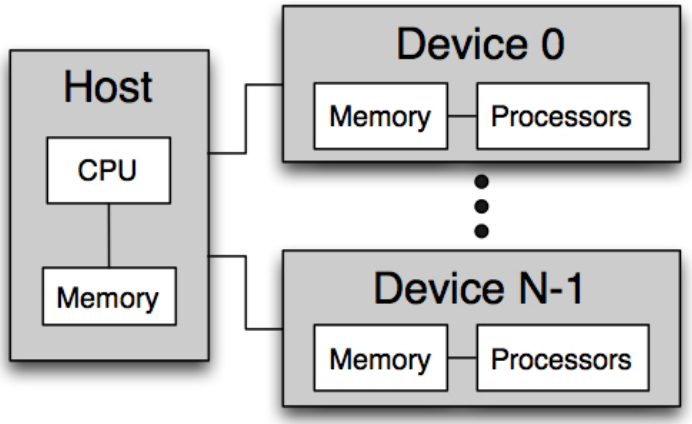
Heterogeneous Programming Library (HPL)

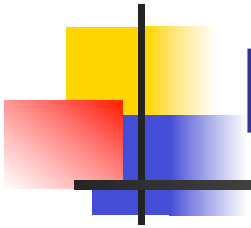


- Facilitates heterogeneous programming
- Based on two elements
  - Kernels: functions that are evaluated in parallel by multiple threads on any device
  - Data type to express arrays and scalars that can be used in kernels and host/serial code
- Implementation based in C++ and OpenCL
  - http://hpl.des.udc.es

HW/SW model

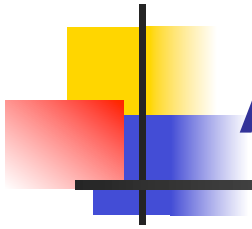
- Serial code runs in the host
- Parallel kernels can be run everywhere
  - Semantics like CUDA and OpenCL
- Processors can only access their memory





Kernels

- Supports kernels
  - In standard OpenCL
  - Developed in an embedded language
- Embedded language has
  - Macros for control structures
  - Predefined variables
  - Functions (sync, arithmetic ops, etc.)



Array data type

- Array<type, n [,memoryFlag]> defines a n-dimensional array that can be used in host code and kernels
  - Example: Array<float, 2> mx(100, 100)
  - n=0 defines a scalar
  - memoryFlag defines the kind of memory (global, local, constant, or private)

Example: SAXPY (Y=*a*X+Y)

Host-side Arrays

Uses own host storage

Uses existing host storage

```
#include "HPL.h"

using namespace HPL;

float myvector[1000];

Array<float, 1> x(1000), y(1000, myvector);

void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
  y[idx] = a * x[idx] + y[idx];
}

int main() {
  float a;

  //the vectors are filled in with data (not shown)

  eval(saxpy)(y, x, a);
}
```

Example: SAXPY (Y=*a**X+Y)

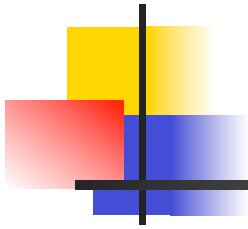
Kernel
Idx = global thread id

```
#include "HPL.h"

using namespace HPL;

float myvector[1000];

Array<float, 1> x(1000), y(1000, myvector);

void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
   y[idx] = a * x[idx] + y[idx];
}

int main() {
  float a;

  //the vectors are filled in with data (not shown)

  eval(saxpy)(y, x, a);
}
```

Example: SAXPY (Y=*a**X+Y)

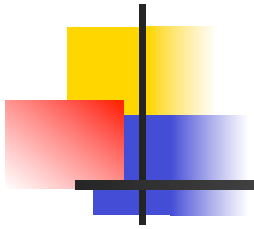
```
#include "HPL.h"

using namespace HPL;

float myvector[1000];

Array<float, 1> x(1000), y(1000, myvector);

void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
   y[idx] = a * x[idx] + y[idx];
}

int main() {
  float a;

  //the vectors are filled in with data (not shown)

  eval(saxpy)(y, x, a);
}
```

Request kernel execution



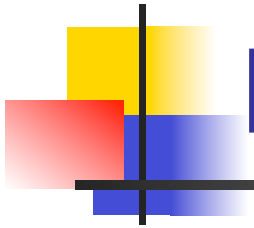
HTA + HPL integration

- I. Data type integration
  - Simplify the joint usage of the HTA and Array types
- II. Coherency management
  - Guarantee HTA and HPL invocations use valid versions of data



I. Data type integration

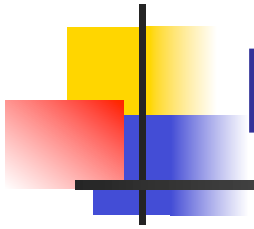
- HTAs are global multi-node objects, while HPL Arrays are per-node structures
  - Solution: definition of Arrays associated to the local HTA tiles
- Typical HTA pattern: A single tile per node identified by the process/node id
- Build HPL Arrays so that their host-side memory is the one of the HTA tile



Example

Gets number of processes/nodes

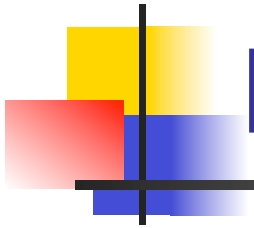
```
int N = Traits::Default::nPlaces();

auto h_arr = HTA<float, 2>::alloc({100, 100}, {N, 1});

int MYID = Traits::Default::myPlace();

Array<float, 2> local_arr(100, 100, h_arr({MYID, 1}).raw());
```



Example

Build an HTA with a column on N tiles of size 100x100
(each tile is placed in a different node)

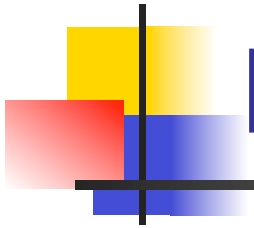
```
int N = Traits::Default::nPlaces();

auto h_arr = HTA<float, 2>::alloc({100, 100}, {N, 1});

int MYID = Traits::Default::myPlace();

Array<float, 2> local_arr(100, 100, h_arr({MYID, 1}).raw());
```



Example

Get id of the local node

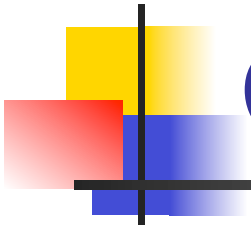
```
int N = Traits::Default::nPlaces();

auto h_arr = HTA<float, 2>::alloc({100, 100}, {N, 1});

int MYID = Traits::Default::myPlace();

Array<float, 2> local_arr(100, 100, h_arr({MYID, 1}).raw());
```



Example

Build local HPL Array of 100x100 elements whose host-side memory is in the place of the local tile of the HTA

```
int N = Traits::Default::nPlaces();

auto h_arr = HTA<float, 2>::alloc({100, 100}, {N, 1});

int MYID = Traits::Default::myPlace();

Array<float, 2> local_arr(100, 100, h_arr({MYID, 1}).raw());

/* Rule:
   - Use h_arr     for CPU/internote operations
   - Use local_arr for accelerator   operations
*/
```



Coherency management

- HPL manages the coherency of its Arrays
  - Automated transfers to/from/between devices
- Must know whether an Array was read and/or written in each usage
  - Kernel ops: Known from kernel analysis
  - Host ops: Known from accessor or manually reported by a method called data
- HTA operations are host operations
  - Inform on them to HPL Arrays using the data API

Example

Place(s) where each array is updated

Matrix A	Matrix B	Matrix C
Host	Host	Host

```
auto hta_A = HTA<float,2>::alloc({{(HA/N), WA}, {N, 1}});
Array<float,2> lA((HA/N), WA, hta_A({MY_ID}).raw());

auto hta_B = HTA<float,2>::alloc({{(HB/N), WB}, {N, 1}});
Array<float,2> lB((HB/N), WB, hta_B({MY_ID}).raw());

auto hta_C = HTA<float,2>::alloc({{HC, WC}, {N, 1}});
Array<float,2> lC(HC, WC, hta_C({MY_ID}).raw());

hta_A = 0.f;
eval(fillinB)(lB);
hmap(fillinC, hta_C);
eval(mxmul)(lA, lB, lC, HC, alpha);

lA.data(HPL_RD); // Brings A data to the host
double result = hta_A.reduce(plus<double>());
```

Example

Place(s) where each array is updated

Matrix A	Matrix B	Matrix C
Host	Host	Host
Host	Host	Host

```
auto hta_A = HTA<float,2>::alloc({{(HA/N), WA}, {N, 1}});
Array<float,2> lA((HA/N), WA, hta_A({MY_ID}).raw());

auto hta_B = HTA<float,2>::alloc({{(HB/N), WB}, {N, 1}});
Array<float,2> lB((HB/N), WB, hta_B({MY_ID}).raw());

auto hta_C = HTA<float,2>::alloc({{HC, WC}, {N, 1}});
Array<float,2> lC(HC, WC, hta_C({MY_ID}).raw());

hta_A = 0.f;
eval(fillinB)(lB);
hmap(fillinC, hta_C);
eval(mxmul)(lA, lB, lC, HC, alpha);

lA.data(HPL_RD); // Brings A data to the host
double result = hta_A.reduce(plus<double>());
```

Example

Place(s) where each array is updated

Matrix A	Matrix B	Matrix C
Host	Host	Host
Host	Host	Host
Host	GPU	Host

```
auto hta_A = HTA<float,2>::alloc({{(HA/N), WA}, {N, 1}});
Array<float,2> lA((HA/N), WA, hta_A({MY_ID}).raw());

auto hta_B = HTA<float,2>::alloc({{(HB/N), WB}, {N, 1}});
Array<float,2> lB((HB/N), WB, hta_B({MY_ID}).raw());

auto hta_C = HTA<float,2>::alloc({{HC, WC}, {N, 1}});
Array<float,2> lC(HC, WC, hta_C({MY_ID}).raw());

hta_A = 0.f;
eval(fillinB)(lB);
hmap(fillinC, hta_C);
eval(mxmul)(lA, lB, lC, HC, alpha);

lA.data(HPL_RD); // Brings A data to the host
double result = hta_A.reduce(plus<double>());
```

Example

Place(s) where each array is updated

Matrix A	Matrix B	Matrix C
Host	Host	Host
Host	Host	Host
Host	GPU	Host
Host	GPU	Host

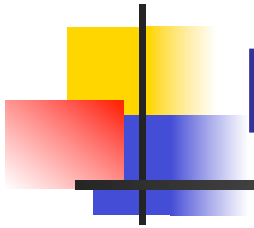
```
auto hta_A = HTA<float,2>::alloc({{(HA/N), WA}, {N, 1}});
Array<float,2> lA((HA/N), WA, hta_A({MY_ID}).raw());

auto hta_B = HTA<float,2>::alloc({{(HB/N), WB}, {N, 1}});
Array<float,2> lB((HB/N), WB, hta_B({MY_ID}).raw());

auto hta_C = HTA<float,2>::alloc({{HC, WC}, {N, 1}});
Array<float,2> lC(HC, WC, hta_C({MY_ID}).raw());

hta_A = 0.f;
eval(fillinB)(lB);
hmap(fillinC, hta_C);
eval(mxmul)(lA, lB, lC, HC, alpha);

lA.data(HPL_RD); // Brings A data to the host
double result = hta_A.reduce(plus<double>());
```

Example

Place(s) where each array is updated

Matrix A	Matrix B	Matrix C
Host	Host	Host
Host	Host	Host
Host	GPU	Host
Host	GPU	Host
GPU	GPU	Both

```
auto hta_A = HTA<float,2>::alloc({{(HA/N), WA}, {N, 1}});
Array<float,2> lA((HA/N), WA, hta_A({MY_ID}).raw());

auto hta_B = HTA<float,2>::alloc({{(HB/N), WB}, {N, 1}});
Array<float,2> lB((HB/N), WB, hta_B({MY_ID}).raw());

auto hta_C = HTA<float,2>::alloc({{HC, WC}, {N, 1}});
Array<float,2> lC(HC, WC, hta_C({MY_ID}).raw());

hta_A = 0.f;
eval(fillinB)(lB);
hmap(fillinC, hta_C);
eval(mxmul)(lA, lB, lC, HC, alpha);

lA.data(HPL_RD); // Brings A data to the host
double result = hta_A.reduce(plus<double>());
```

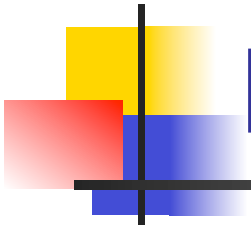


Example

Place(s) where each array is updated

Matrix A	Matrix B	Matrix C
Host	Host	Host
Host	Host	Host
Host	GPU	Host
Host	GPU	Host
GPU	GPU	Both
Both	GPU	Both

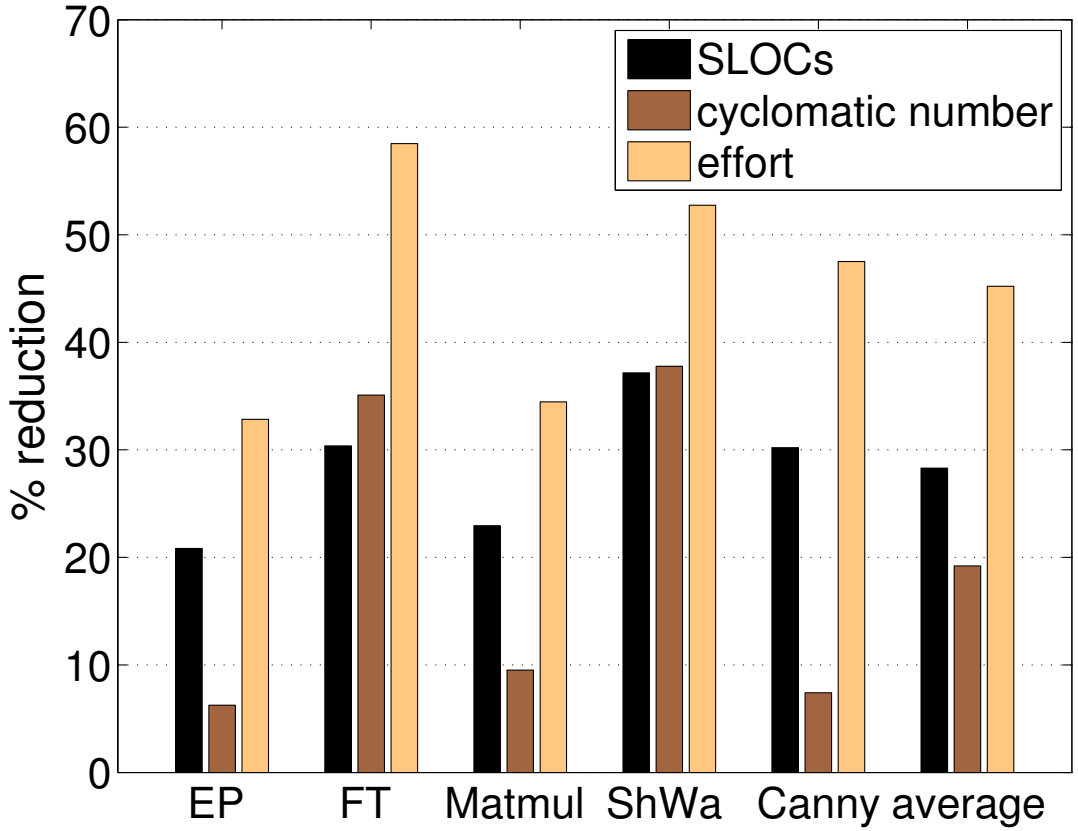
```
auto hta_A = HTA<float,2>::alloc({{(HA/N), WA}, {N, 1}});
Array<float,2> lA((HA/N), WA, hta_A({MY_ID}).raw());

auto hta_B = HTA<float,2>::alloc({{(HB/N), WB}, {N, 1}});
Array<float,2> lB((HB/N), WB, hta_B({MY_ID}).raw());

auto hta_C = HTA<float,2>::alloc({{HC, WC}, {N, 1}});
Array<float,2> lC(HC, WC, hta_C({MY_ID}).raw());

hta_A = 0.f;
eval(fillinB)(lB);
hmap(fillinC, hta_C);
eval(mxmul)(lA, lB, lC, HC, alpha);

lA.data(HPL_RD); // Brings A data to the host
double result = hta_A.reduce(plus<double>());
```

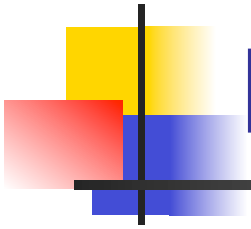


Evaluation

- Applications:
  - EP: Embarrassingly parallel with reduction
  - FT: FFT with all-to-all communications
  - Matmul: Distributed matrix product
  - ShWa: finite-volume scheme (repetitive stencil)
  - Canny: Finds edges in images (stencil)

Programmability versus MPI+OpenCL

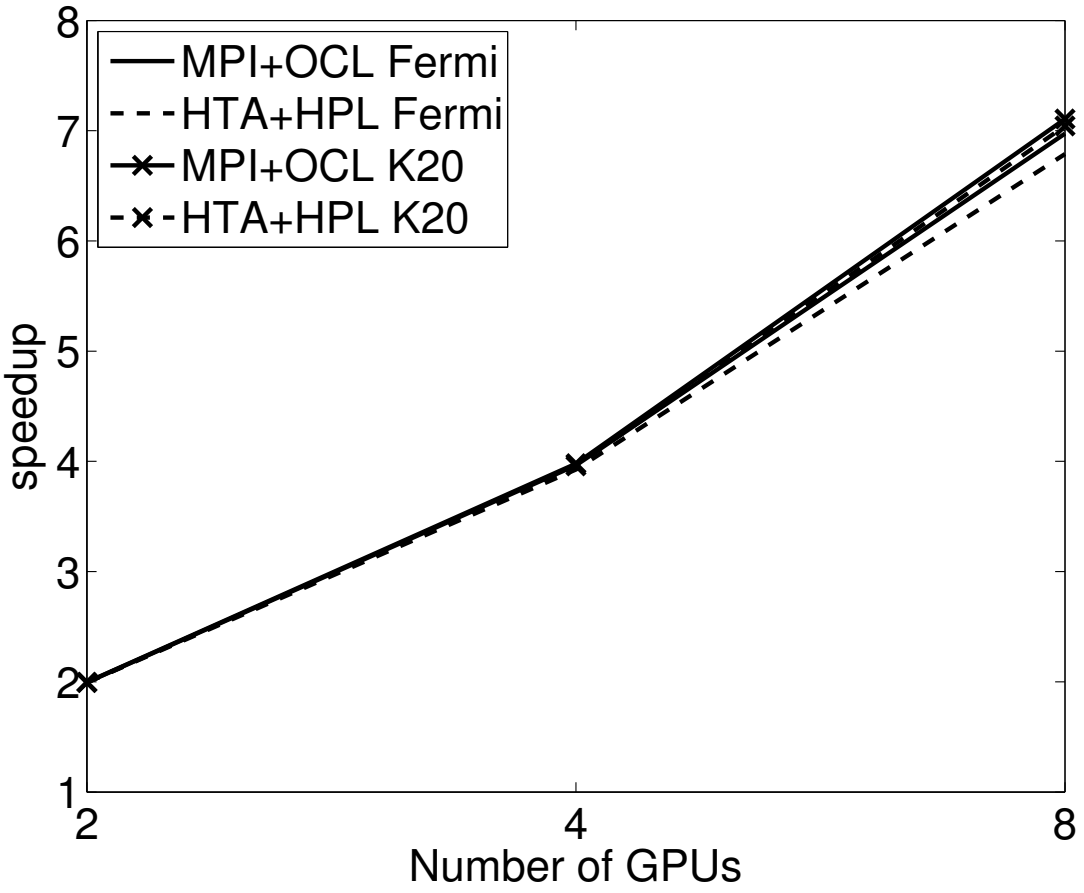




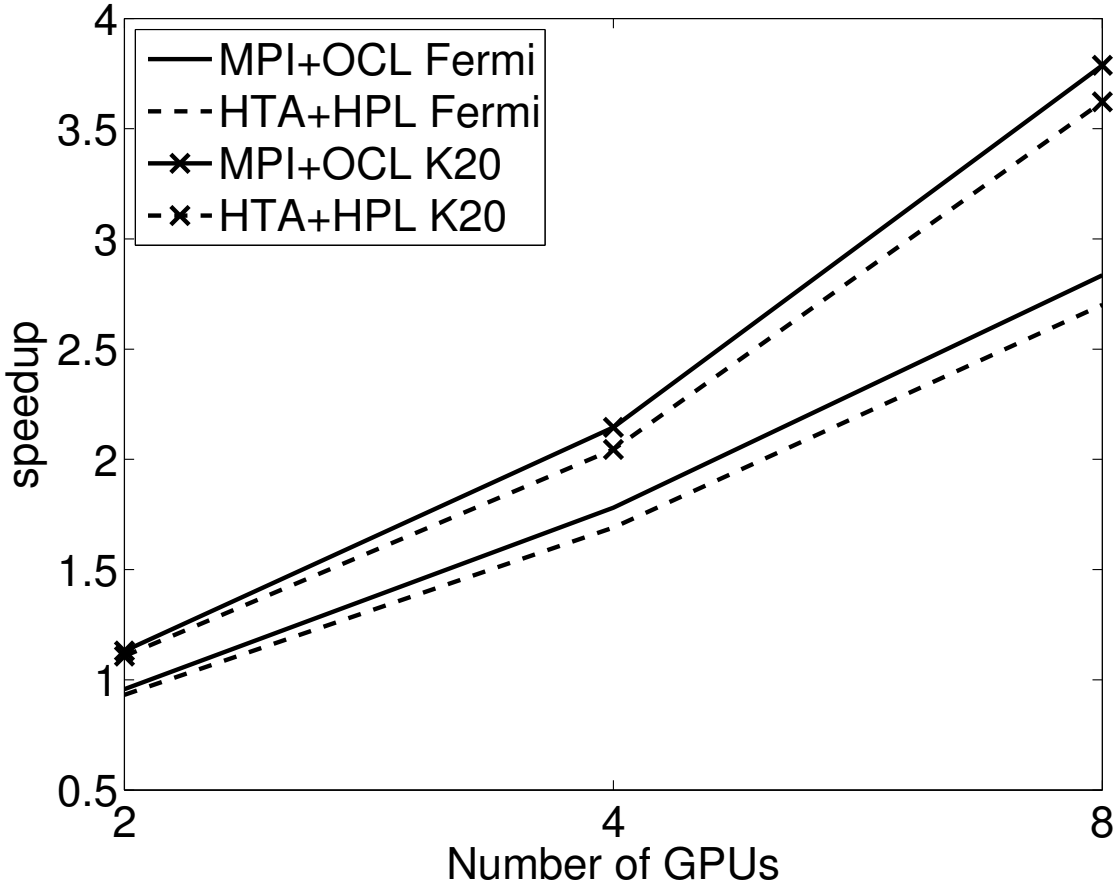
Performance evaluation

- Fermi cluster: 4 nodes with an Intel Xeon X5650 with 6 cores, 12 GB, and 2 Nvidia M2050 GPUs with 3GB each
- K20 cluster: 8 nodes with two Intel Xeon E5-2660 8-core CPUs, 64 GB, and a K20m GPU with 5 GB
- g++ 4.7.2 with optimization level O3

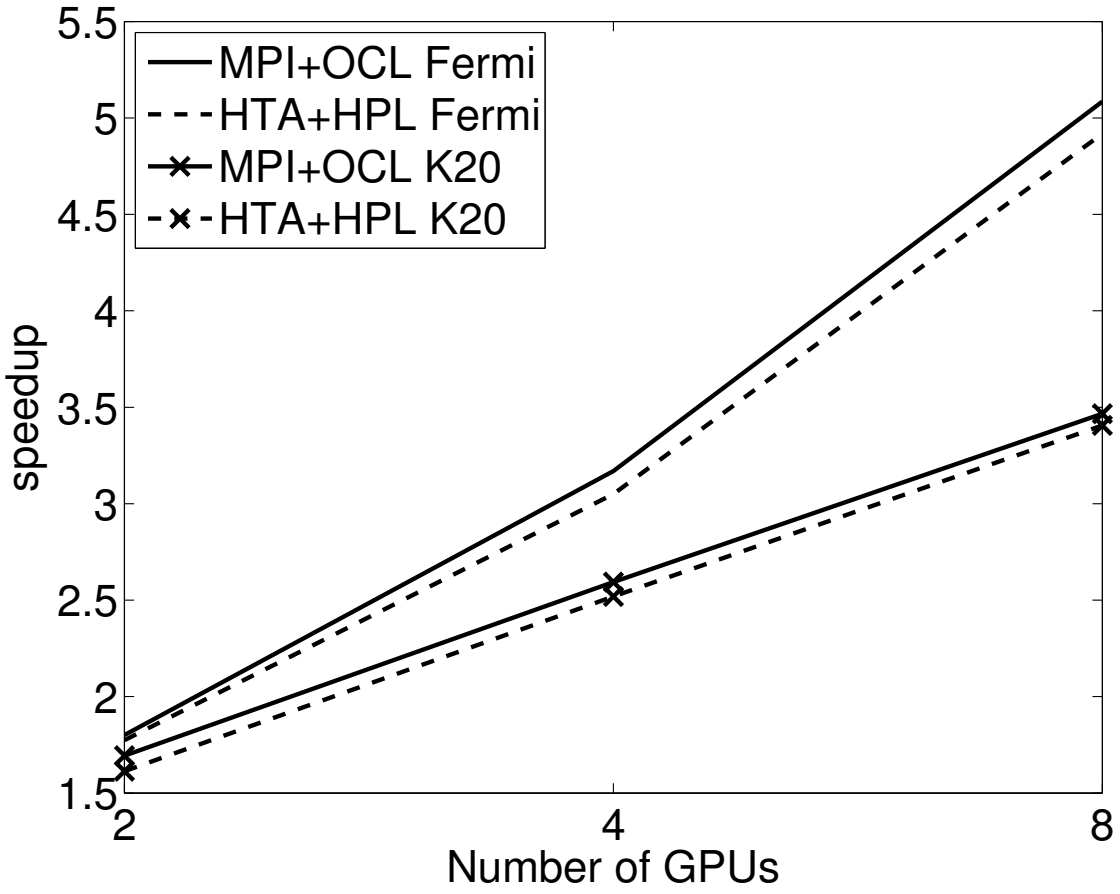
Performance for EP

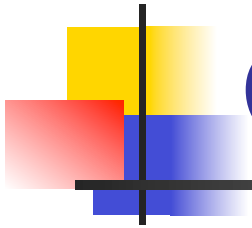


Performance for FT



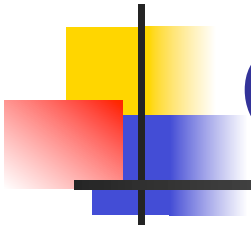
Performance for ShWa





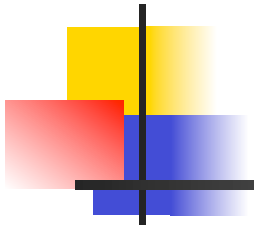
Conclusions & future work

- Heterogeneous clusters are notoriously difficult to program
- Most proposals to improve the situation still require noticeable effort
- We explored combining
  - distributed arrays with global semantics provided by HTAs
  - simpler heterogeneous programming provided by HPL



Conclusions & future work

- Average programmability improvements w.r.t MPI+OpenCL between 19% and 45% (peak of 58%)
- Average overhead just around 2%
- Future work: integrate both tools into a single one



Thank you