OpenMP 4.0 Device Support in the OMPi Compiler

Alexandros Papadogiannakis
Spiros N. Agathos
Vassilios V. Dimakopoulos

Department of Computer Science & Engineering
UNIVERSITY OF IOANNINA
Ioannina, Greece

IWOMP 2015
30.09 - 02.10 - GERMANY

IKY
ΙΔΡΥΜΑ ΚΡΑΤΙΚΩΝ ΥΠΟΤΡΟΦΙΩΝ
STATE SCHOLARSHIPS FOUNDATION

Presentation Overview

- Area of Interest
- OpenMP Device Model
- Designing OpenMP device support in OMPi
  - Compiler Transformations
  - General runtime Architecture
  - Device data environments
- Prototype implementation for the Parallella
  - Experimental Results

OpenMP Device Model (1/3)

- OpenMP v4.0 of specification
  - SIMD constructs to vectorize serial/parallelized loops
  - Thread affinity
  - Conditional cancelation of parallel regions
  - Tasking extensions
    - Task dependencies
    - Task grouping
  - **Device Support**
    - **Create data environment in the device**
    - **Instruct device to execute code regions (kernel)**

OpenMP Device Model (3/3)

- Current state of the art
  - GCC
  - ICC

  } Xeon Phi

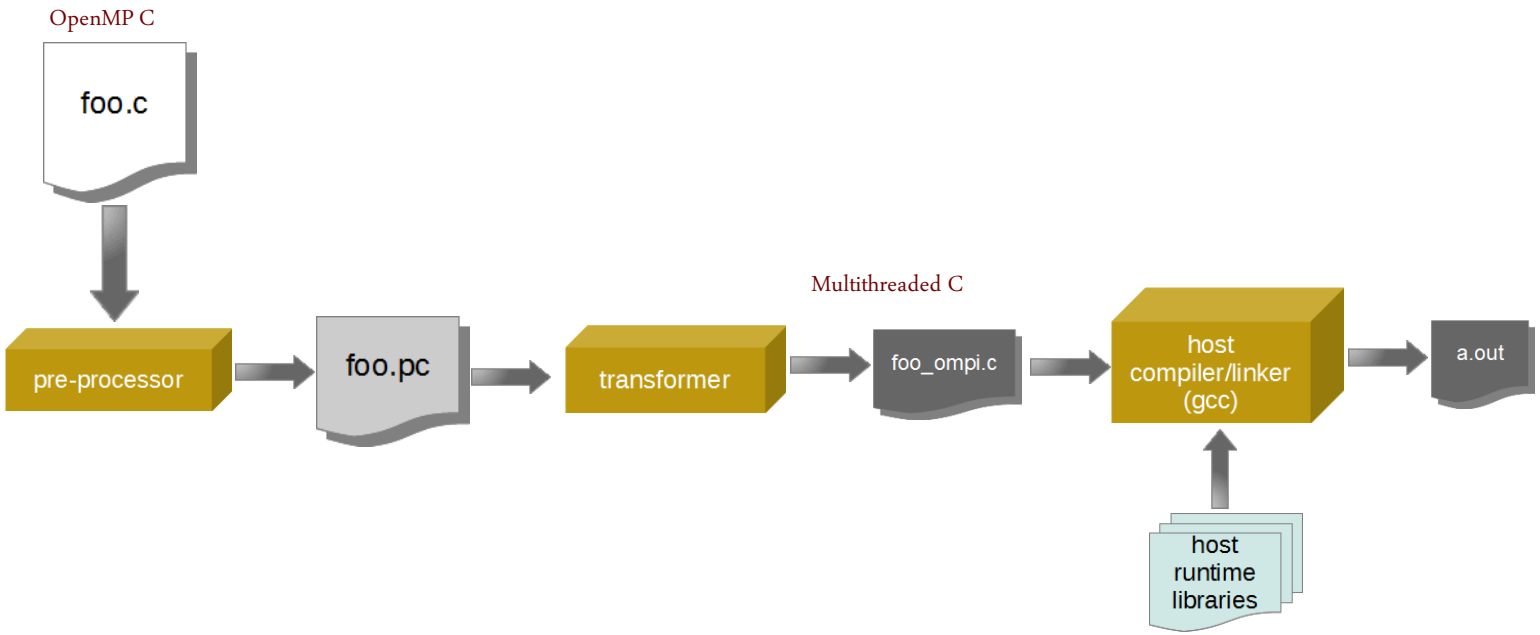
- Preliminary support for the device model
  - Rose compiler
  - LLVM

- OMPi : Goal is to provide multi device support

Designing OpenMP device support in OMPi

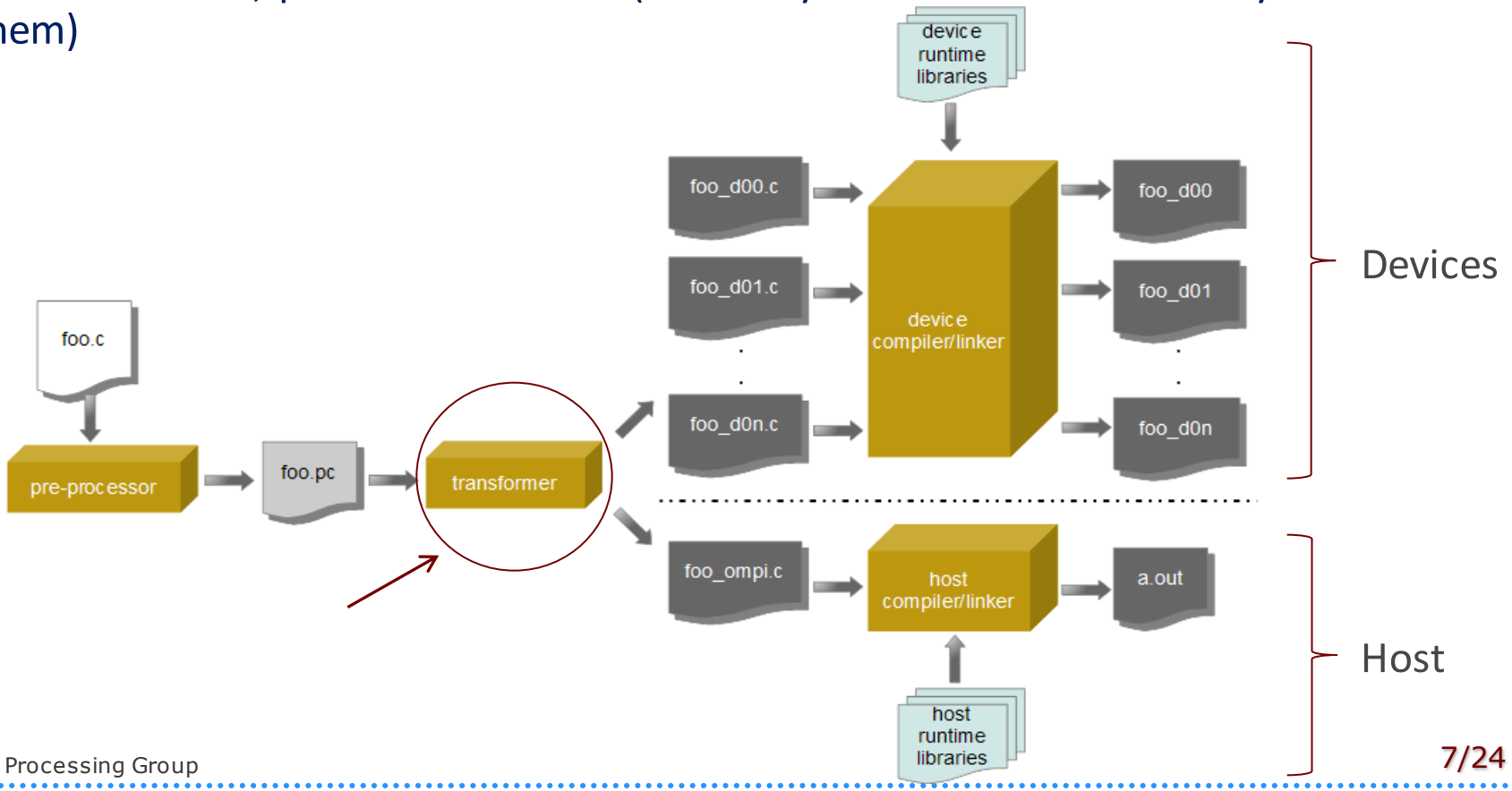
Designing OpenMP device support

- OMPi (http://paragroup.cs.uoi.gr/wpsite/software/ompi):
  - V3.1 OpenMP C infrastructure
  - Source to source compiler + Runtime libraries



Compiler Transformations (1/3)

- Input grammar modified with the new target-related directives
- AST augmented with new nodes to represent the new directives
- Each kernel (i.e. target region), is outlined to a separate function
- The code generation phase produces multiple output files, one for each different kernel, plus the Host code (Host may be called to execute any of them)



Compiler Transformations (3/3)

OpenMP original code :

```
int x, y, z;

#pragma omp target data map(x,y)
   #pragma omp target map(from: x) device (2)
      x=y=z=1;
```

- Specifications allow arbitrary nesting of directives of different devices
  - 1st environment → default device
  - 2nd environment → specified device
- Default device may change
- Compiler is unable to handle the data environments
- Our solution: Delegate this to the runtime

Compiler Transformations (3/3)

OpenMP original code :

```
int x, y, z;

#pragma omp target data map(x,y)
   #pragma omp target map(from: x) device (2)
      x=y=z=1;
```

```
{ // start target data
  _devid = -1; // default device
  _ddenv = _start_ddenv(NULL , _devid , ..);

  _end_ddenv(_ddenv );
}
```

Compiler Transformations (3/3)

OpenMP original code :

```
int x, y, z;

#pragma omp target data map(x,y)
   #pragma omp target map(from: x) device (2)
      x=y=z=1;
```

```
{ // start target data
  _devid = -1; // default device
  _ddenv = _start_ddenv(NULL , _devid , ..);

  _initvar (&x, sizeof(x), _ddenv , _devid );
  _initvar (&y, ..);

  _finvar (&x, _ddenv );
  _finvar (&y, _ddenv );

  _end_ddenv(_ddenv );
}
```

Compiler Transformations (3/3)

OpenMP original code :

```
int x, y, z;

#pragma omp target data map(x,y)
   #pragma omp target map(from: x) device (2)
      x=y=z=1;
```

```
{ // start target data
  _devid = -1; // default device
  _ddenv = _start_ddenv(NULL , _devid , ..);
  _initvar (&x, sizeof(x), _ddenv , _devid );
  _initvar (&y, ..);

{ // start target
   _devid = 2; // requested device
   _ddenv_prev = _ddenv;
   _ddenv = _start_ddenv(_ddenv_prev , _devid , ..);


   _end_ddenv(_ddenv );
 } // end target

  _finvar (&x, _ddenv );
  _finvar (&y, _ddenv );
  _end_ddenv(_ddenv );
}
```

Compiler Transformations (3/3)

OpenMP original code :

```
int x, y, z;

#pragma omp target data map(x,y)
   #pragma omp target map(from: x) device (2)
      x=y=z=1;
```

```
{ // start target data
  _devid = -1; // default device
  _ddenv = _start_ddenv(NULL , _devid , ..);
  _initvar (&x, sizeof(x), _ddenv , _devid );
  _initvar (&y, ..);

  { // start target
    _devid = 2; // requested device
    _ddenv_prev = _ddenv;
    _ddenv = _start_ddenv(_ddenv_prev , _devid , ..);

    _allocvar (&x, ..);
    _initvar (&y, ..);

    _finvar (&x, _ddenv );
    _finvar (&y, _ddenv );

    _end_ddenv(_ddenv );
  } // end target

  _finvar (&x, _ddenv );
  _finvar (&y, _ddenv );
  _end_ddenv(_ddenv );
}
```

Compiler Transformations (3/3)

OpenMP original code :

```
int x, y, z;

#pragma omp target data map(x,y)
   #pragma omp target map(from: x) device (2)
      x=y=z=1;
```

```
{ // start target data
  _devid = -1; // default device
  _ddenv = _start_ddenv(NULL , _devid , ..);
  _initvar (&x, sizeof(x), _ddenv , _devid );
  _initvar (&y, ..);

  { // start target


struct __dd__ {
 int (* x);
 int (* y);
 int z;
 } *_devdata = _devdata_alloc(_devid ,
                sizeof(struct __dd__ ));

_devdata ->x = get_vaddress (&x, ..);
_devdata ->y = get_vaddress (&y, ..);
_devdata ->z = z; // optimized

ort_offload_kernel(_kernelFunc0_ , _devdata , ..);

z = _devdata ->z;


  } // end target

  _finvar (&x, _ddenv );
  _finvar (&y, _ddenv );
  _end_ddenv(_ddenv );
}
```

Compiler Transformations (3/3)

OpenMP original code :

```
int x, y, z;

#pragma omp target data map(x,y)
   #pragma omp target map(from: x) device (2)
      x=y=z=1;
```

```
{ // start target data
  _devid = -1; // default device
  _ddenv = _start_ddenv(NULL , _devid , ..);
  _initvar (&x, sizeof(x), _ddenv , _devid );
  _initvar (&y, ..);

  { // start target
    _devid = 2; // requested device
    _ddenv_prev = _ddenv;
    _ddenv = _start_ddenv(_ddenv_prev , _devid , ..);

    _allocvar (&x, ..); // ignored if default device is 2
    _initvar (&y, ..);  // ditto

    struct __dd__ {
      int (* x);
      int (* y);
      int z;
    } *_devdata = _devdata_alloc(_devid , sizeof(struct __dd__ ));

    _devdata ->x = get_vaddress (&x, ..); // request address @device
    _devdata ->y = get_vaddress (&y, ..);
    _devdata ->z = z; // optimized

    ort_offload_kernel(_kernelFunc0_ , _devdata , ..); // kernel code

    z = _devdata ->z;

    _finvar (&x, _ddenv );
    _finvar (&y, _ddenv );

    _end_ddenv(_ddenv );
  } // end target

  _finvar (&x, _ddenv );
  _finvar (&y, _ddenv );
  _end_ddenv(_ddenv );
}
```

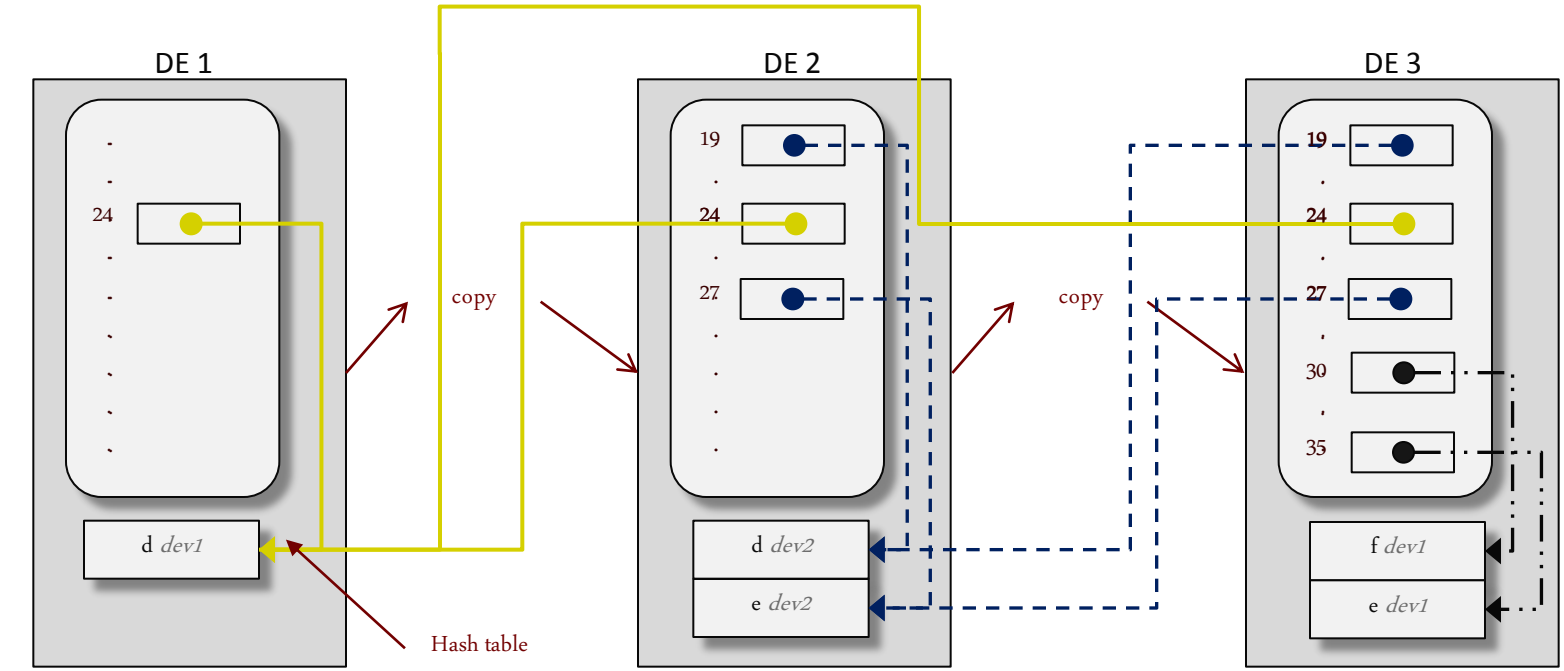
Handling device data environments in the runtime (1/4)

- The bookkeeping cannot be statically handled at compile time
- We utilize a novel bookkeeping mechanism:
  - Separately-chained hash tables (ht)
  - Operated by the runtime system, resides in the Host address space
  - Each data environment spawns a new hash table

Handling device data environments in the runtime (2/4)

```
int d, e, f;

#pragma omp target data map(d) device (1)          // DE1
   #pragma omp target data map(d, e) device (2)    // DE2
      #pragma omp target map(f) device (1)         // DE3
      {
         e = d++; // implicit mapping of variable e
              ... // in device 1
      }
```



target data map(d) device(1)

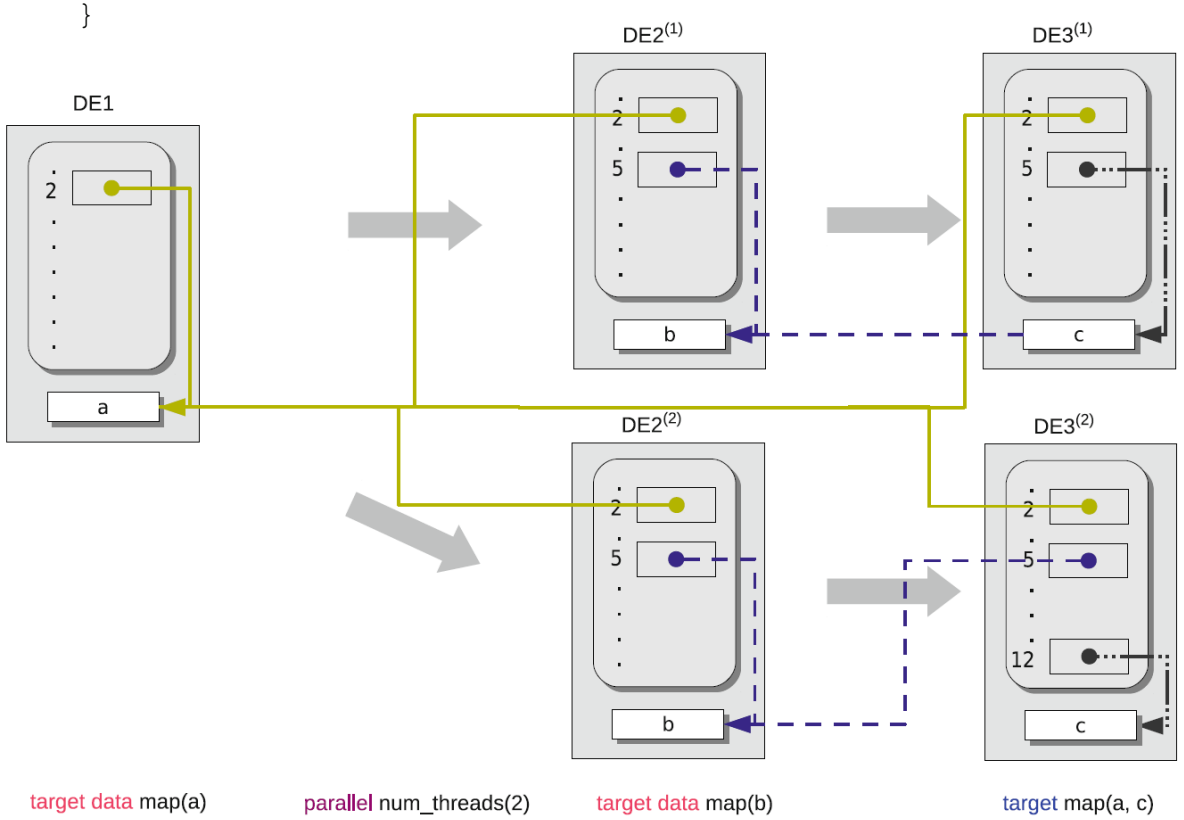
target data map(d, e) device(2)

target map(f) device(1)
[implicit map(e)]

Handling device data environments in the runtime (3/4)

```
int a, b;

#pragma omp target data map(a)              // DE1
   #pragma omp parallel num_threads(2)
   {
      int c;
      #pragma omp target data map(b)        // DE2^(1) & DE2^(2)
      #pragma omp target map(a, c)          // DE3^(1) & DE3^(2)
        …
   }
```

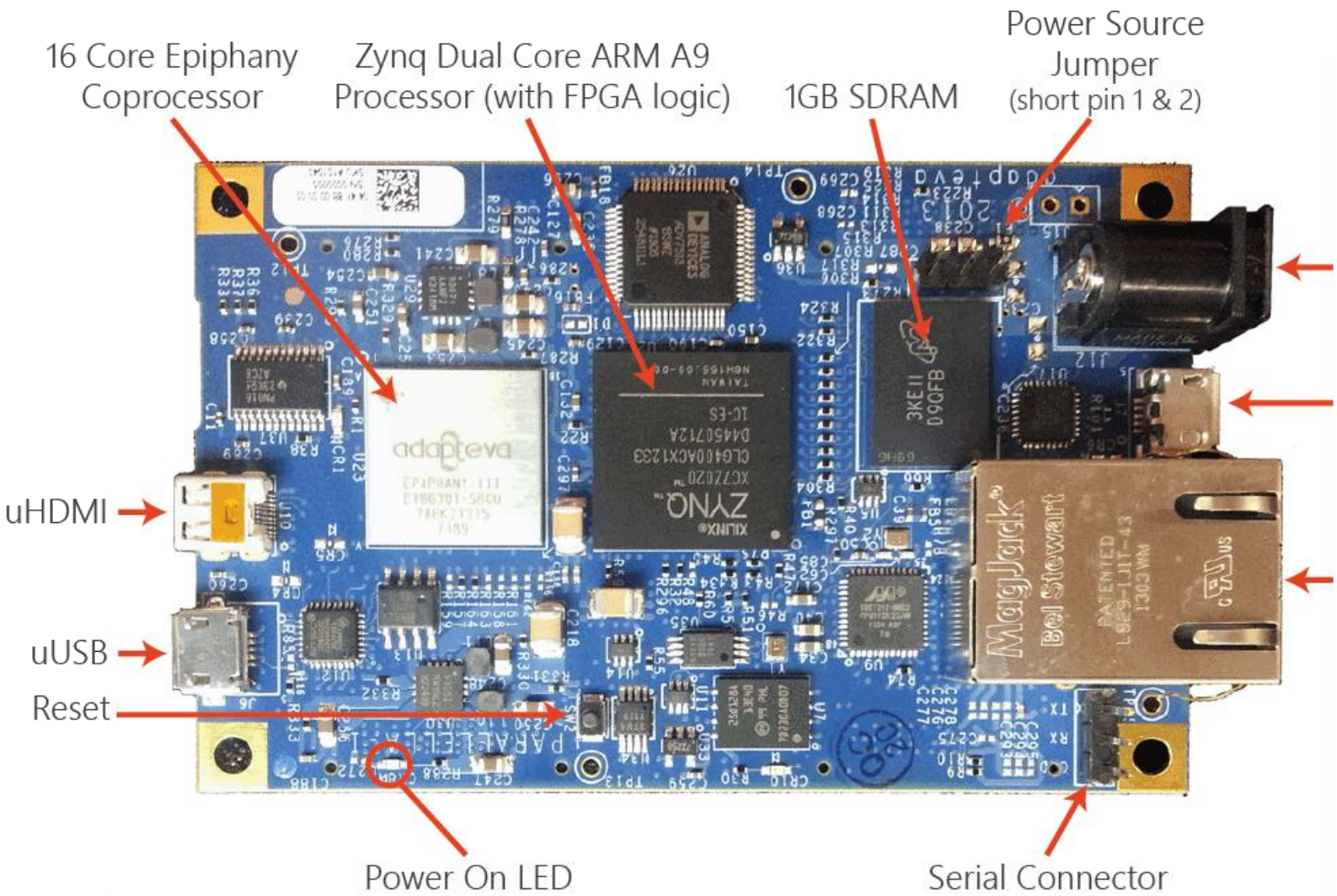


Handling device data environments in the runtime (4/4)

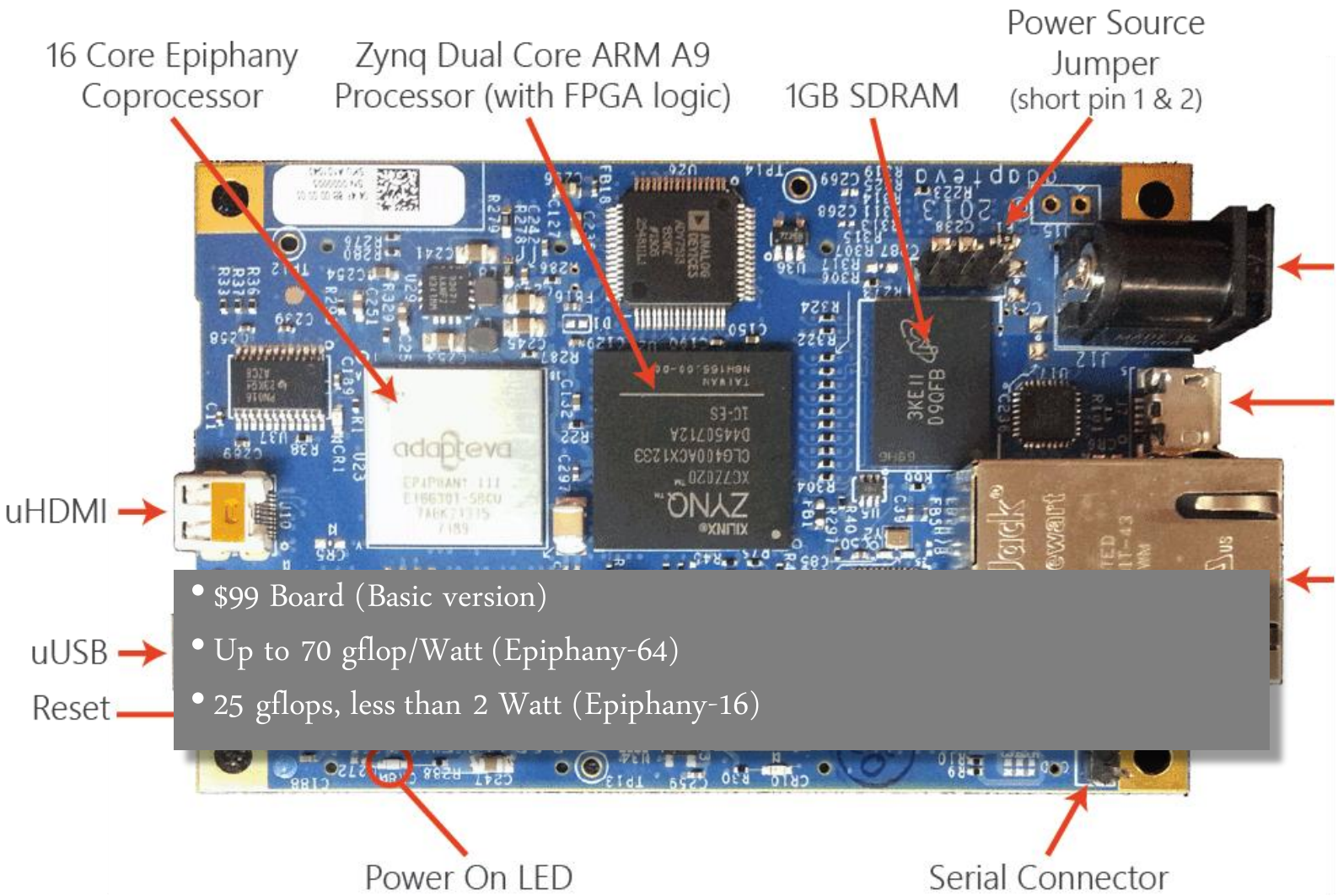
- Space requirements: **O(L · K + n)**
  - *L* is the maximum number of alive data environments
  - *K* is the size of a hash table
  - *n* is the total number of variables that are mapped in all data environment definitions
- Number of memory allocations and de-allocations: **Θ(E)**
  - *E* is the total number of data environments

Prototype implementation for the Parallella

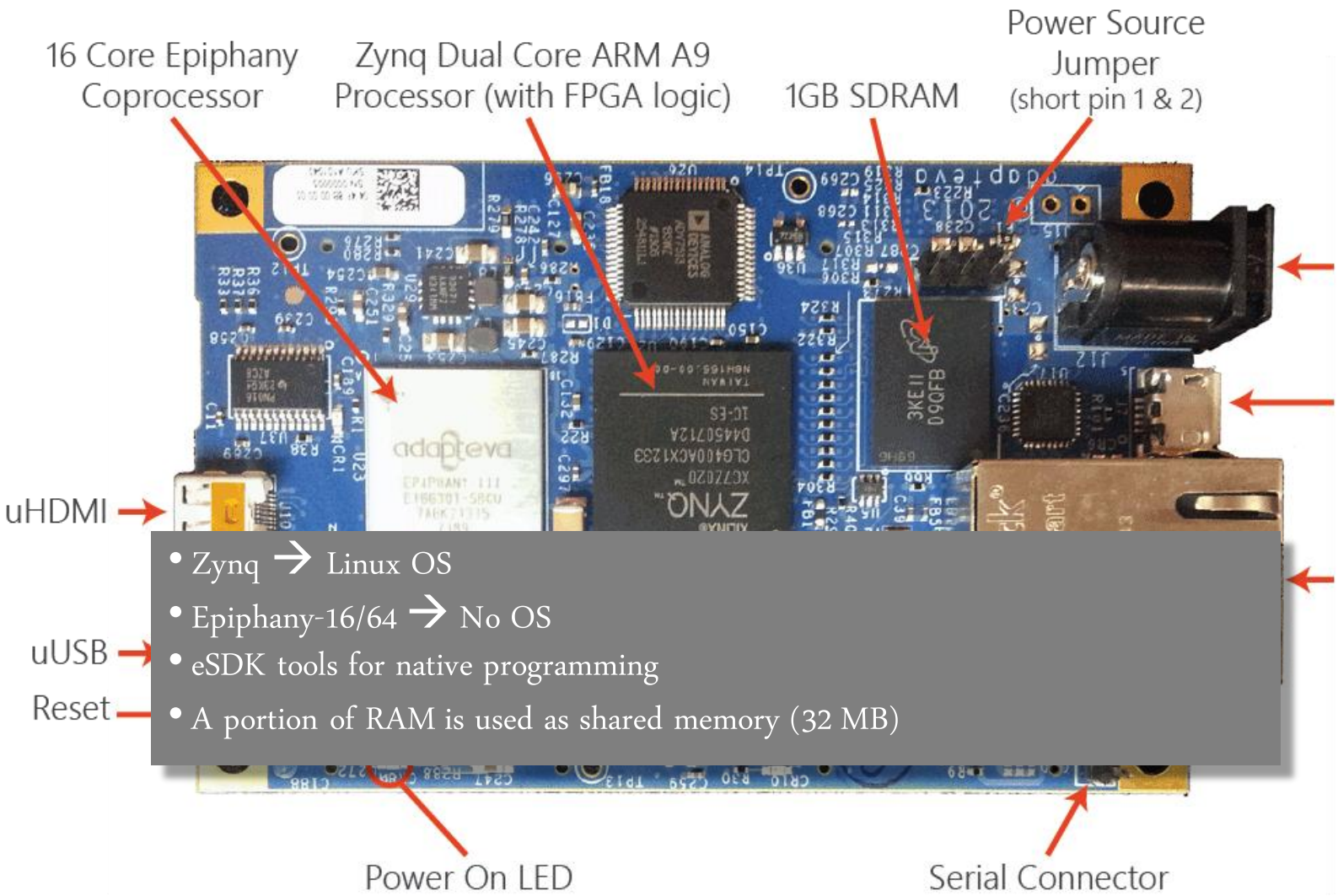
Parallella Board



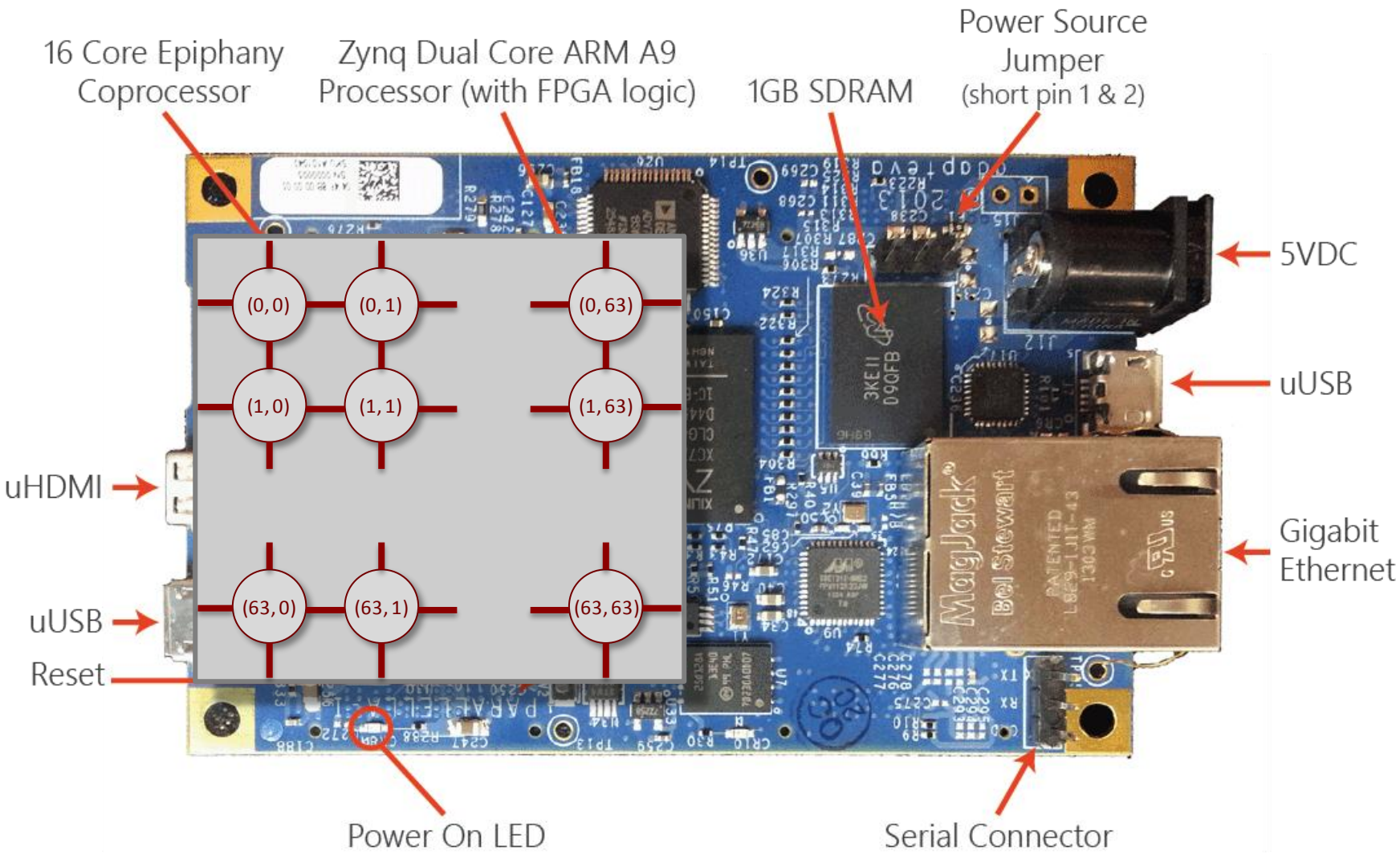
Parallella Board



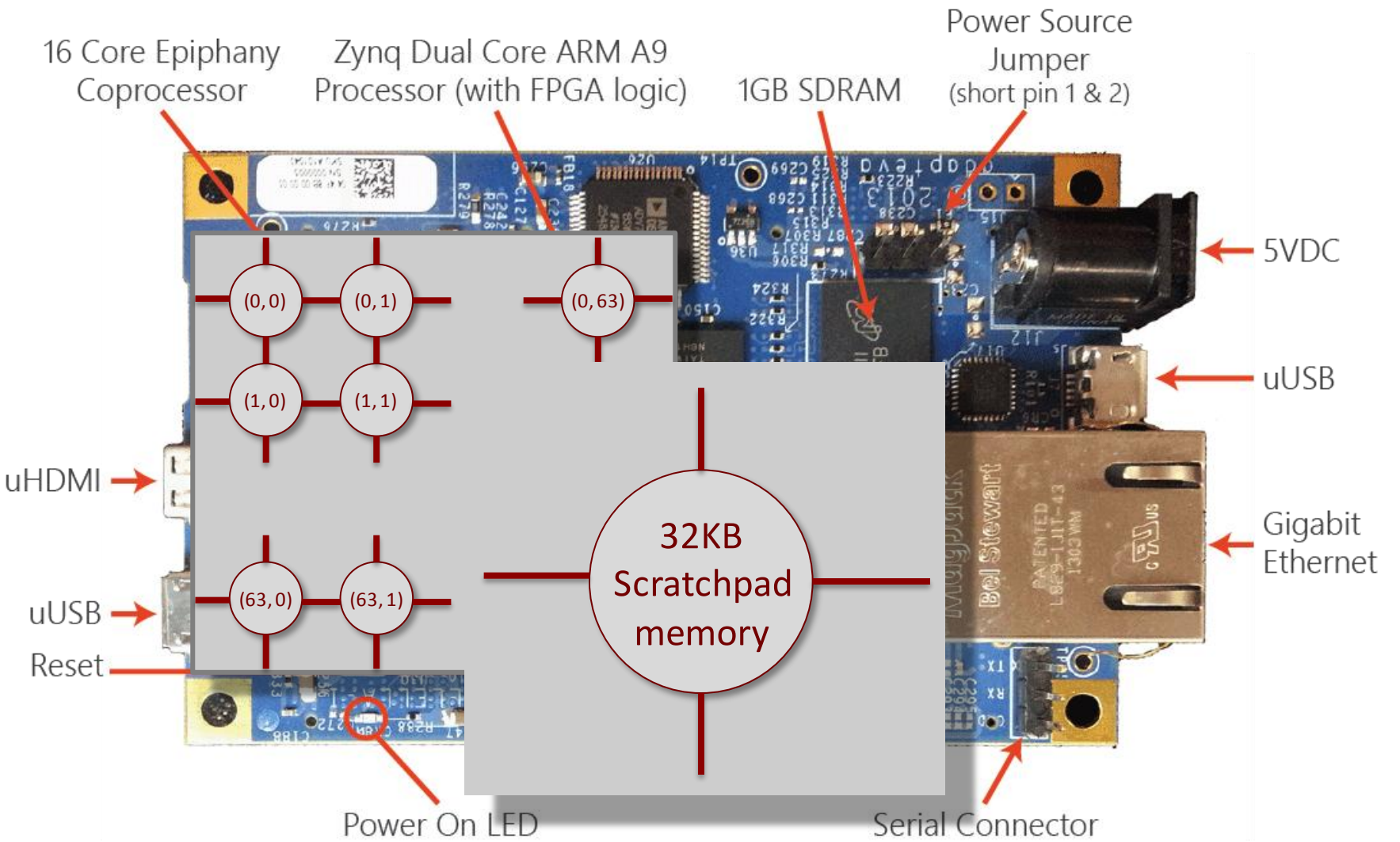
Parallella Board



Parallella Board

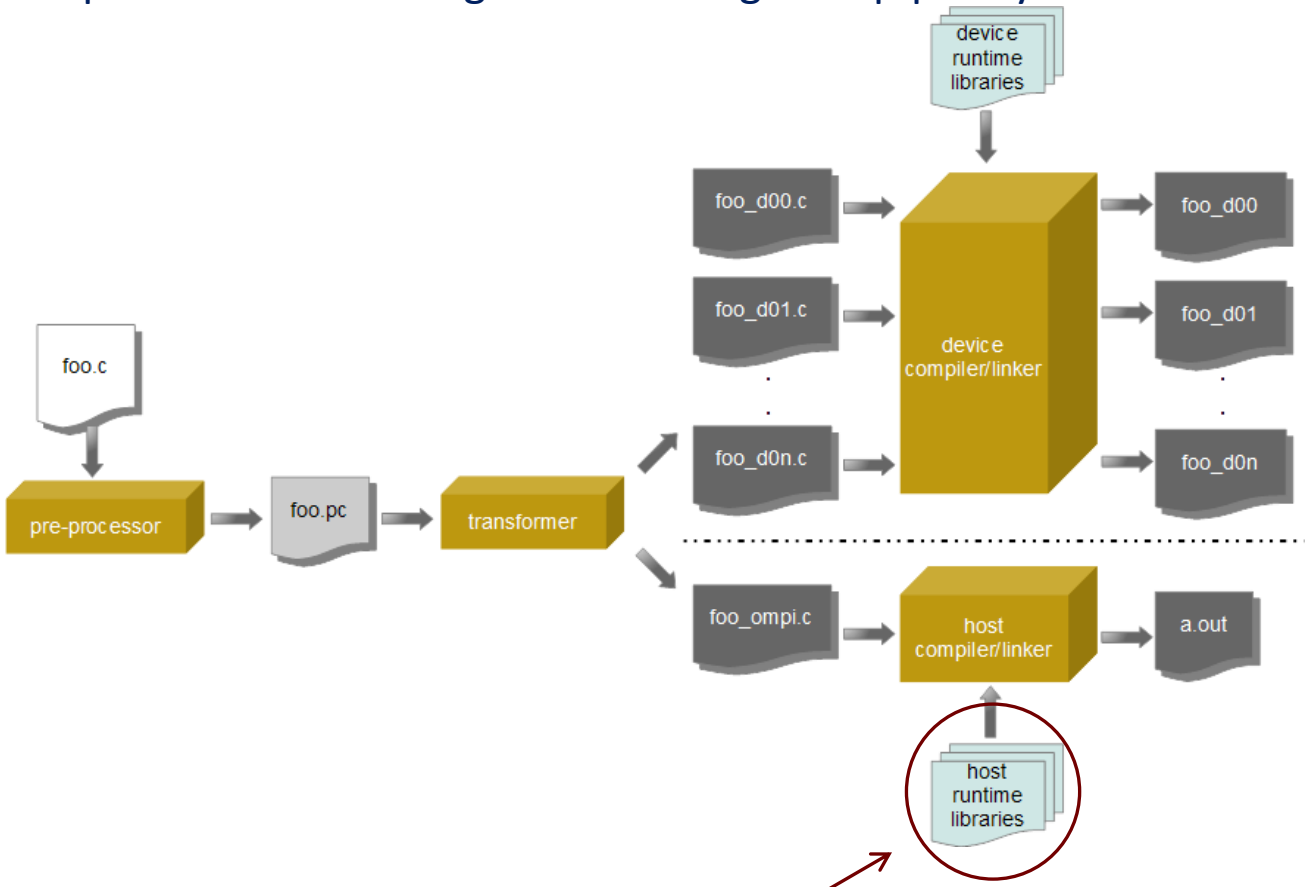


Parallella Board



Runtime Architecture - What the Host does

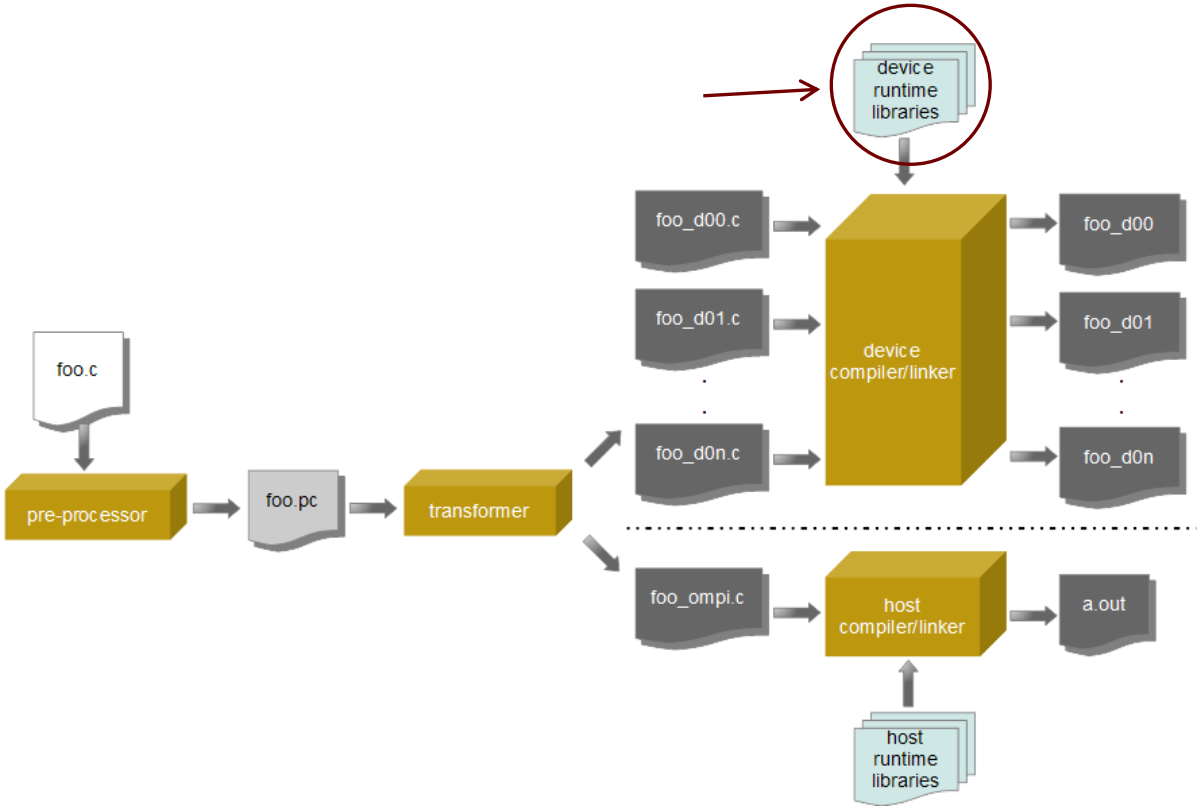
- A full-fledged OpenMP runtime library
  - Supporting execution on the dual-ARM processor
- Additional functionality
  - Required for controlling and accessing the Epiphany device



Runtime Architecture - What the Host does

- A full-fledged OpenMP runtime library
  - Supporting execution on the dual-ARM processor
- Additional functionality
  - Required for controlling and accessing the Epiphany device
- The communication between the Host and the eCores takes place through the shared memory portion of the system RAM
- For offloading a kernel
  - The first idle eCore is chosen
  - Precompiled object file is loaded to it for immediate execution
  - Ecores inform the Host about the completion of a kernel through special flags in shared memory.
  - Multiple Host threads can offload multiple independent kernels concurrently onto the Epiphany

OpenMP Within the Epiphany



OpenMP Within the Epiphany

- Difficult to support OpenMP within the Epiphany
  - ECores do not execute any OS
  - No provision for dynamic parallelism
  - The 32KiB local memory is quite limited:
    - Unable to handle sophisticated OpenMP runtime structures
- The runtime infrastructure originally designed for the Host was trimmed down to a minimum
  - This is linked and offloaded with each kernel

OpenMP Within the Epiphany

- Only the Host can activate the eCores, to provide dynamic parallelism:
  - The master core contacts the Host, requesting the activation of a number of cores
  - A copy of the same kernel is then offloaded to the newly activated cores
  - Master and workers synchronize to execute a parallel region
  - The corresponding coordination among the participating eCores utilizes the local memory of the team's master eCore

- Prototype tasking infrastructure
  - Based on a blocking shared queue
  - Stored in the local memory of the master eCore
  - The corresponding task data environments are stored in the shared memory

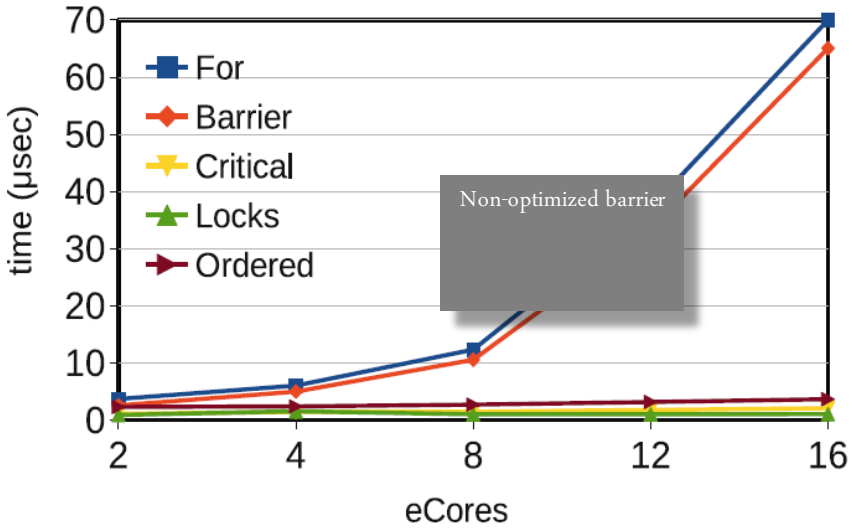
Experimental Results

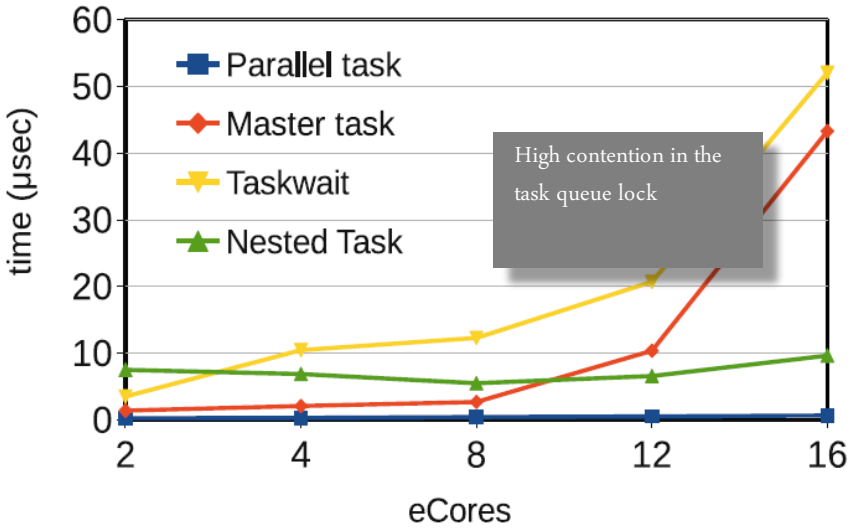
Experimental Results (1/3)

- Environment
  - Parallella-16 SKUA101020
  - Ubuntu 14.04, kernel 3.12.0 armv7l
  - gcc and e-gcc v.4.8.2 as back-end for OMPi
  - eSDK 5.13.9.10

Experimental Results (2/3)

- Modified version of the EPCC benchmarks
  - Basic routines are offloaded through target directives
  - Measurements from the Host side after subtracting any offloading costs



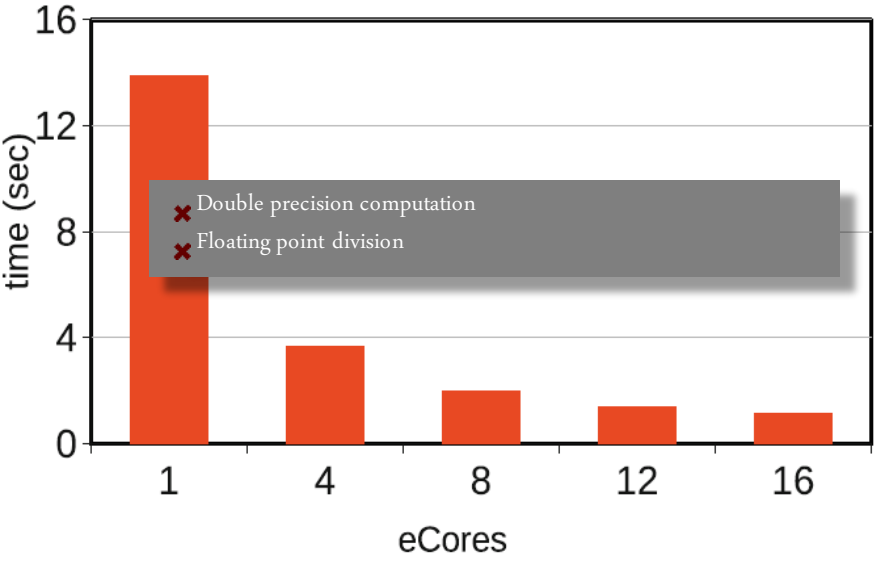


Overhead results of EPCC benchmark executed on the Epiphany-16 through offloading

Experimental Results (3/3)

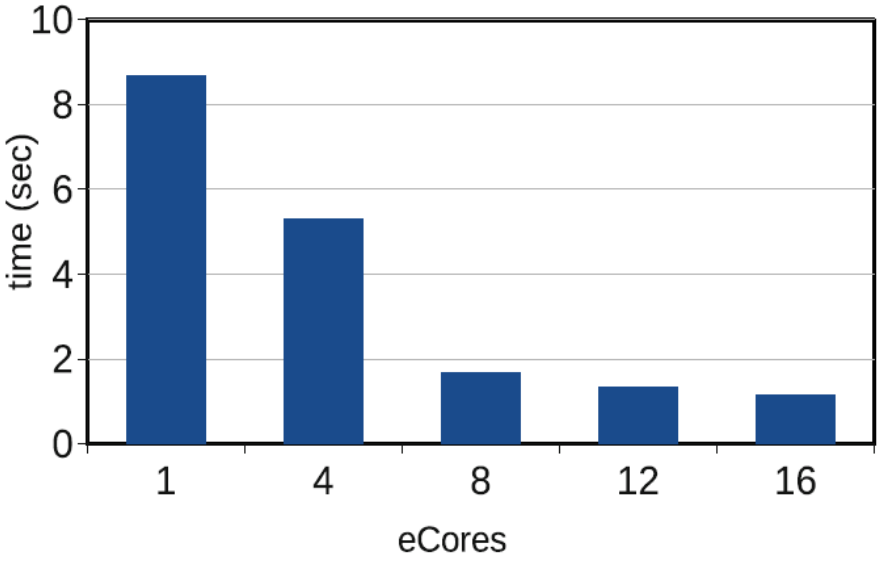
Pi computation

2.000.000 intervals



Nqueens(12)

Cut-off version – 144 tasks



Conclusion…

- The experimentation with the Parallella board showed:
  - Offloaded kernels should not make use of sophisticated OpenMP features, in devices with limited resources
  - Time needed to offload a kernel → major overhead → applications should not repeatedly offload kernels
- A kernel should be offloaded only once & there should be provisions for asynchronous communication between the Host and devices

Current status

- We are currently testing
  - Cancelation
  - Task dependencies
- Further optimization of the Epiphany runtime libraries
- Adding support for other devices

END...

Thank you

Acknowledgements:

IKY
ΙΔΡΥΜΑ ΚΡΑΤΙΚΩΝ ΥΠΟΤΡΟΦΙΩΝ
STATE SCHOLARSHIPS FOUNDATION

adapteva