Barcelona Supercomputing Center
Centro Nacional de Supercomputación

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

OmpSs@FPGA

Tutorial – Introduction

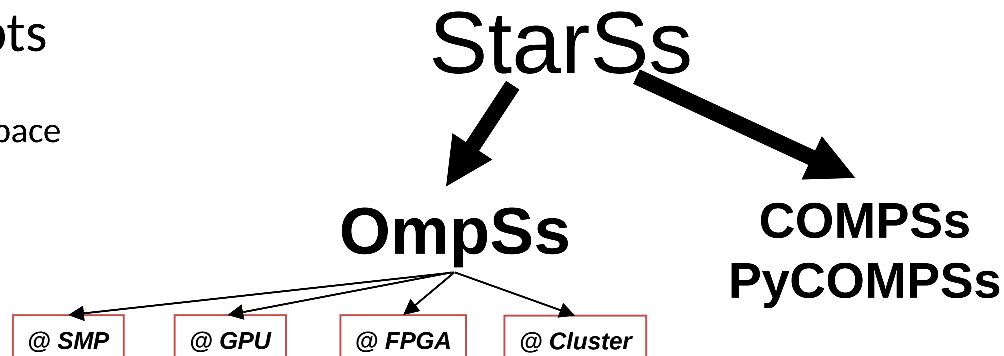
BSC OmpSs@FPGA team

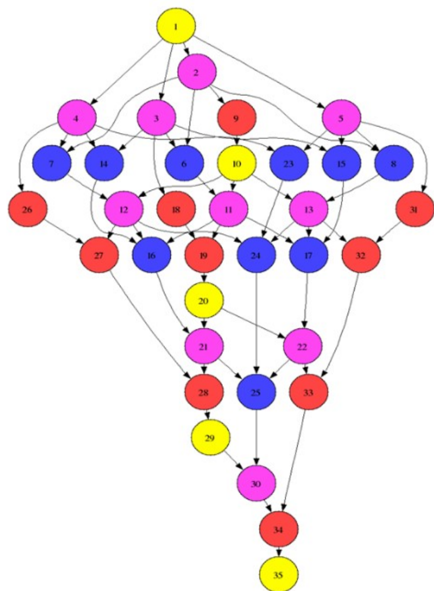
Universitat Politècnica de Catalunya, and
Barcelona Supercomputing Center

November 4th, 2018

Task-based programming

- StarSs family key concepts
  - Sequential task-based program
  - View of a single address/name space
  - Execution in parallel: automatic
  - runtime computation of dependencies
- Productivity and portability



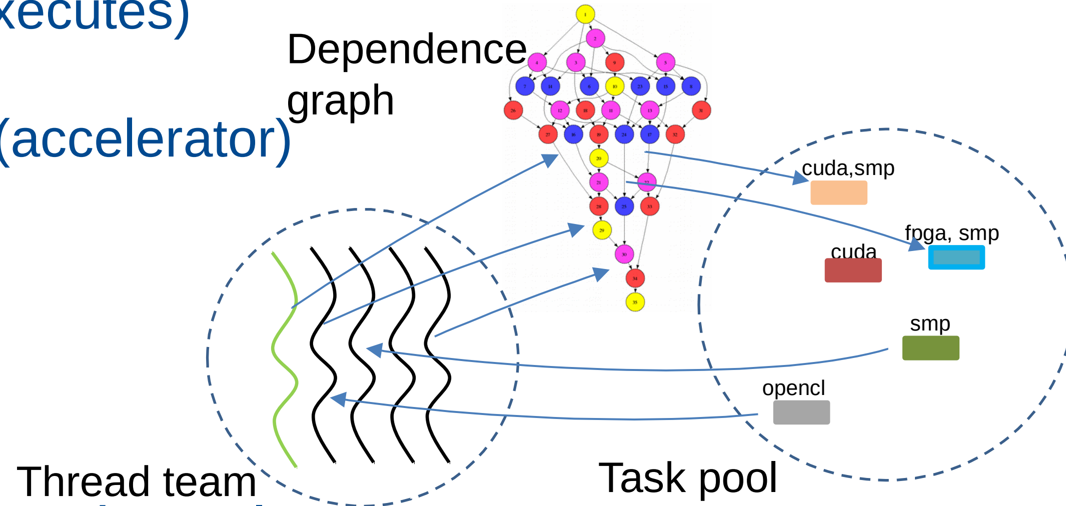


- OmpSs is used for prototyping extensions to OpenMP
  - Tasking
  - data-dependences
  - Heterogeneity
  - Multiple address spaces
  - …
- Mercurium compiler
- Nanos runtime system

Execution Model (OmpSs + accelerators)

Global thread team created on startup

- Master starts main task (also executes)
- N workers execute tasks
- One representative per device (accelerator)



All threads get work from a task pool

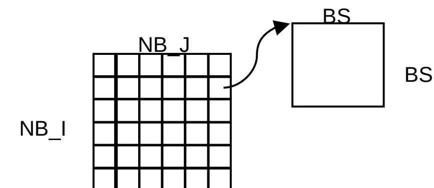
- Accelerator kernels become tasks
- Tasks are labeled with (at least) one target device
  - » smp (default), cuda, opencl, **fpga**
- Scheduler decides which task to execute
- Tasks may have several targets (implements)

www.bsc.es

Easy to Program

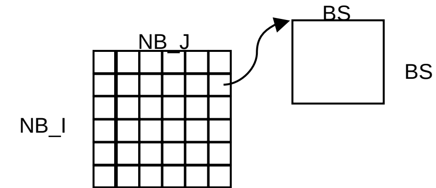
OmpSs Example: Tiled MxM

```
void matrix_multiply(T a[BS][BS],T b[BS][BS],T c[BS][BS]);
…
for (i_b=0; i_b<NB_I; i_b++)
  for (j_b=0; j_b<NB_J; j_b++)
    for (k_b=0; k_b<NB_K; k_b++)
       matrix_multiply(AA[i_b][k_b], BB[k_b][j_b], CC[i_b][j_b]);
…
```



OmpSs Example: Tiled MxM

```
#pragma omp task in([BS]a,[BS]b) inout([BS]c)
void matrix_multiply(T a[BS][BS],T b[BS][BS],T c[BS][BS]);
…
for (i_b=0; i_b<NB_I; i_b++)
  for (j_b=0; j_b<NB_J; j_b++)
    for (k_b=0; k_b<NB_K; k_b++)
       matrix_multiply(AA[i_b][k_b], BB[k_b][j_b], CC[i_b][j_b]);
…
#pragma omp taskwait
// Or
// other tasks depending on input output c of matrix multiply task
```

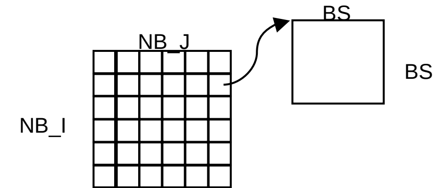


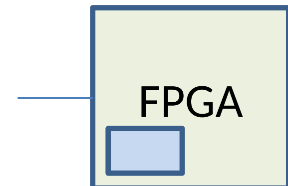
OmpSs Example: Tiled MxM

Mercurium compiler **with autoVivado tool**

- Few extensions (num_instances)
- Triggers the bitstream generation automatically (Stub function generated)

```
#pragma omp target device(fpga) copy_deps num_instances(1)
#pragma omp task in([BS]a,[BS]b) inout([BS]c)
void matrix_multiply(T a[BS][BS],T b[BS][BS],T c[BS][BS]);
…
for (i_b=0; i_b<NB_I; i_b++)
  for (j_b=0; j_b<NB_J; j_b++)
    for (k_b=0; k_b<NB_K; k_b++)
      matrix_multiply(AA[i_b][k_b], BB[k_b][j_b], CC[i_b][j_b]);
…
#pragma omp taskwait
// Or
// other tasks depending on input output c of matrix multiply task
```



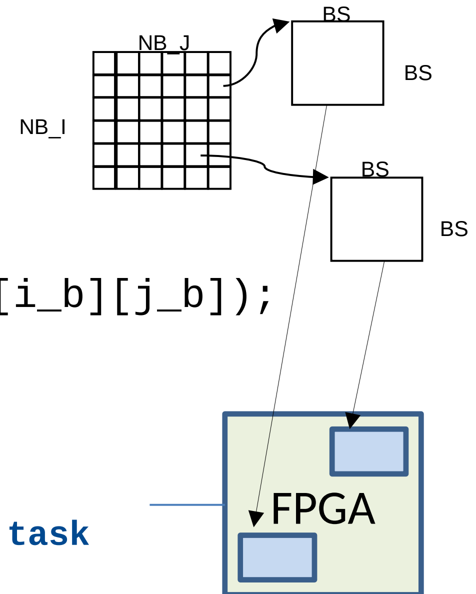


OmpSs Example: Two MxM accelerators

Mercurium compiler **with autoVivado tool**

- Few extensions (num_instances)
- Triggers the bitstream generation automatically (Stub function generated)

```
#pragma omp target device(fpga) copy_deps num_instances(2)
#pragma omp task in([BS]a,[BS]b) inout([BS]c)
void matrix_multiply(T a[BS][BS],T b[BS][BS],T c[BS][BS]);
…
for (i_b=0; i_b<NB_I; i_b++)
  for (j_b=0; j_b<NB_J; j_b++)
    for (k_b=0; k_b<NB_K; k_b++)
       matrix_multiply(AA[i_b][k_b], BB[k_b][j_b], CC[i_b][j_b]);
…
#pragma omp taskwait
// Or
// other tasks depending on input output c of matrix multiply task
```



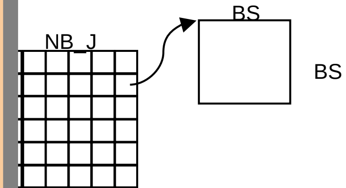
OmpSs Example: MxM kernel + Vivado HLS

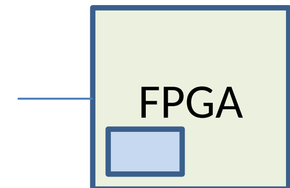
Mercurium compiler **with autoVivado tool**

- Few extensions (num_instances)
- Triggers the bitstream generation automatically (Stub function generated)

```
#pragma omp target device(fpga) copy_deps num_instances(1)
#pragma omp task in([BS]a,[BS]b) inout([BS]c)
void matrix_multiply(T a[BS][BS],T b[BS][BS],T c[BS][BS]);
```

```
#define BS 128
void matrix_multiply(float a[BS][BS], float b[BS][BS],float c[BS][BS])
{
#pragma HLS inline
  int const FACTOR = BS/2;
#pragma HLS array_partition variable=a block factor=FACTOR dim=2
#pragma HLS array_partition variable=b block factor=FACTOR dim=1
  // matrix multiplication of a A*B matrix
  for (int ia = 0; ia < BS; ++ia)
    for (int ib = 0; ib < BS; ++ib) {
#pragma HLS PIPELINE II=1
      float sum = 0;
      for (int id = 0; id < BS; ++id)
        sum += a[ia][id] * b[id][ib];
      c[ia][ib] += sum;
    }}
```



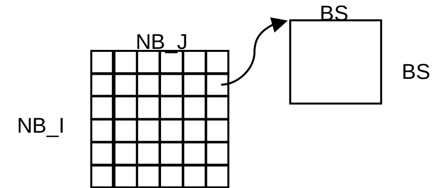


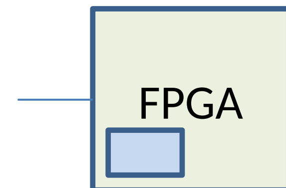
OmpSs Example: MxM kernel + Vivado HLS

Mercurium compiler **with autoVivado tool**

- Few extensions (num_instances)
- Triggers the bitstream generation automatically (Stub function generated)

```
#define BS 128
void matrix_multiply(float a[BS][BS], float b[BS][BS],float c[BS][BS])
{
#pragma HLS inline
  int const FACTOR = BS/2;
#pragma HLS array_partition variable=a block factor=FACTOR dim=2
#pragma HLS array_partition variable=b block factor=FACTOR dim=1
  // matrix multiplication of a A*B matrix
  for (int ia = 0; ia < BS; ++ia)
    for (int ib = 0; ib < BS; ++ib) {
#pragma HLS PIPELINE II=1
      float sum = 0;
      for (int id = 0; id < BS; ++id)
        sum += a[ia][id] * b[id][ib];
      c[ia][ib] += sum;
    }}
```





Xilinx Zynq Support ++

OmpSs FPGA Support: current and future targets...

```
#define BS 128
void matrix_multiply(float a[BS][BS], float b[BS][BS],float c[BS][BS])
{
#pragma HLS inline
  int const FACTOR = BS/2;
#pragma HLS array_partition variable=a block factor=FACTOR dim=2
#pragma HLS array_partition variable=b block factor=FACTOR dim=1
  // matrix multiplication of a A*B matrix
  for (int ia = 0; ia < BS; ++ia)
    for (int ib = 0; ib < BS; ++ib) {
#pragma HLS PIPELINE II=1
      float sum = 0;
      for (int id = 0; id < BS; ++id)
        sum += a[ia][id] * b[id][ib];
      c[ia][ib] += sum;
    }}
```

ZCU102 Board
XCZU9EG-2FFVB1156

Trenz Electronics Zynq U+
TE0808 XCZU9EG-ES1

SECO AXIOM Board
Zynq U+ XCZU9EG-ES2

Zynq-7000 Family
2x Cortex-A9 cores + FPGA (32-bit platforms)

4x Cortex-A53 cores + FPGA (64-bit platforms)

OmpSs Multidimension Example: Simple

```
#pragma omp target device(fpga) copy_deps num_instances(1)
#pragma omp task WHAT SHOULD I WRITE HERE?

void example_mat(T A[32][16][64], T B[32][16][64])
{
...
}

#pragma omp taskwait
```

OmpSs Multidimension Example: Simple

```
#pragma omp target device(fpga) copy_deps num_instances(1)
#pragma omp task in([32]A) out([32]B)

void example_mat(T A[32][16][64], T B[32][16][64])
{
...
}

#pragma omp taskwait
```

www.bsc.es

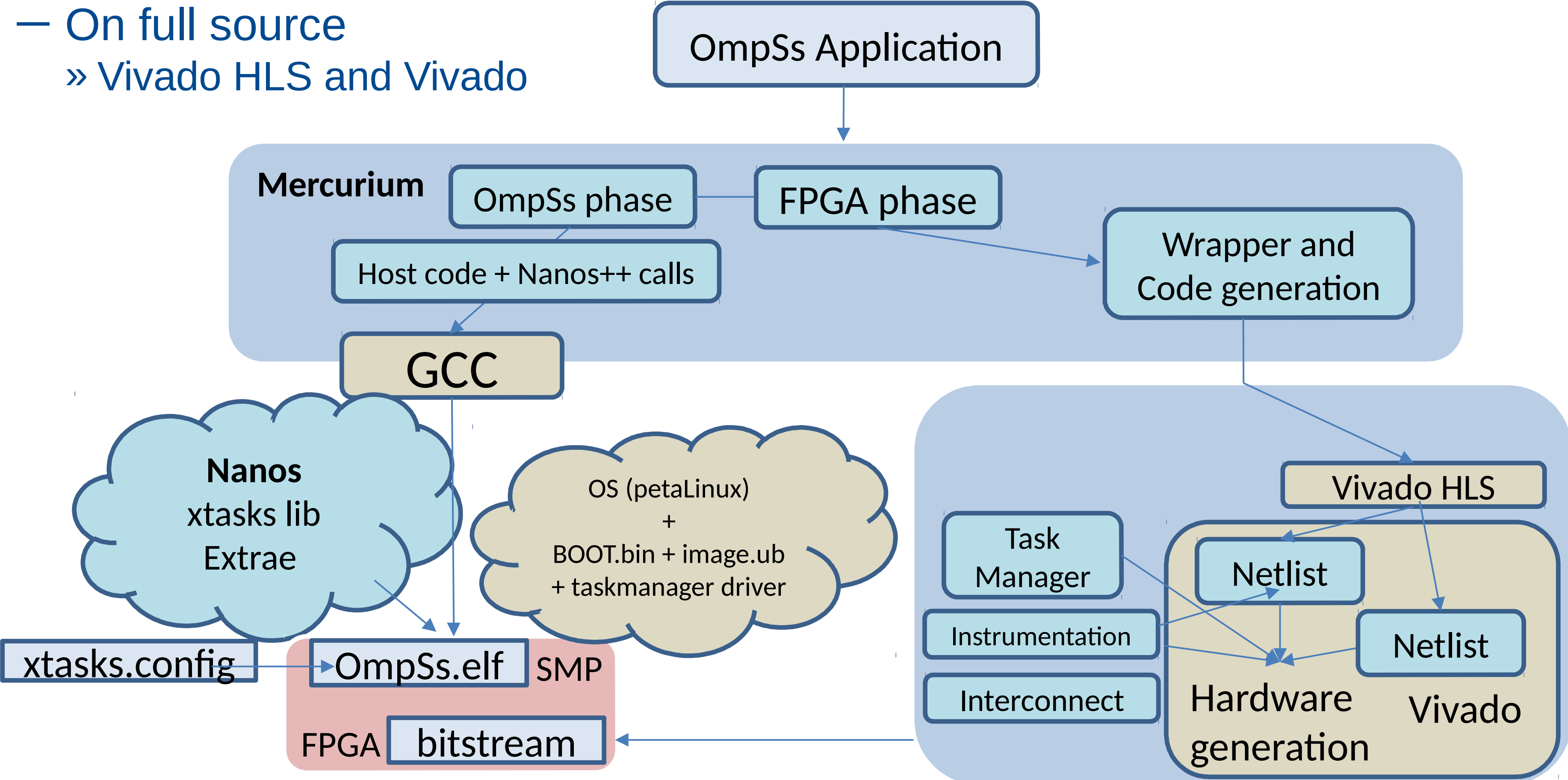
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

OmpSs@FPGA Ecosystem: Integrating FPGA compilation into OmpSs

OmpSs@FPGA Ecosystem: autoVivado

OmpSs transformations

– On full source
  - » Vivado HLS and Vivado



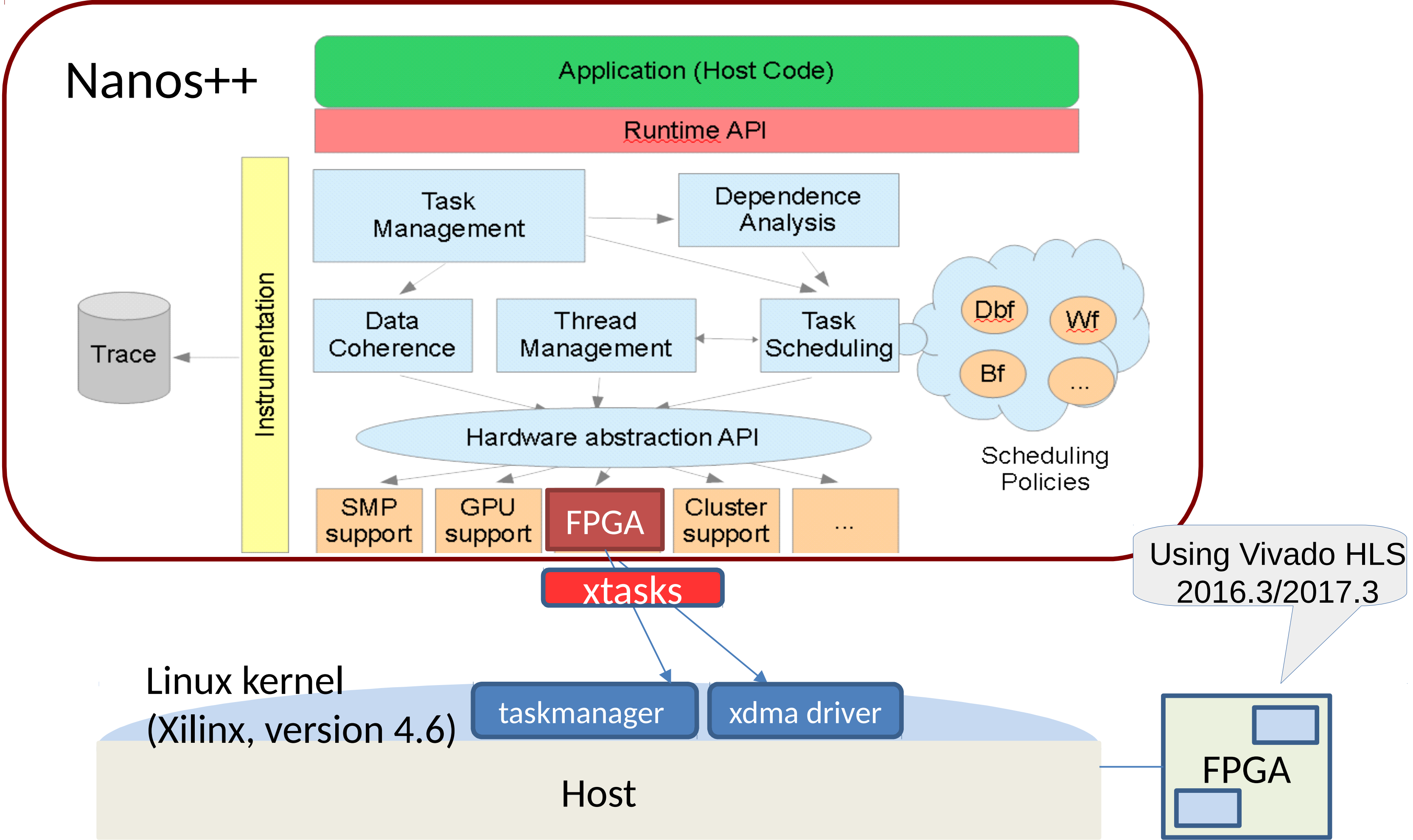
Compilation

Few details on the steps to compile

- Compile: use fpgacc
  - » [cross-compile-]fpgacc --ompss -o program program.c
  - » [cross-compile-]fpgacc --ompss --instrumentation -o program program.c

- Details...
  - » Compiling options:
    - --variable=bitstream_generation:ON
  - » Linking options:
    - --Wf,"board=$(BOARD_NAME),--clock=100,--hardware_instrumentation,--task_manager,--to_step=boot"
    - --Wf,"-v,--name=vivado_project_name,--dir=$(VIVADO_WORKSPACE)"

Execution environment: Nanos++ runtime



Execution

Few details on the steps to run the application

– Environment:
  - Load (insmod) drivers:xmda.ko and taskmanager.ko
  - Download the bitstream:
    - Zynq 7000 family: cat program.bin > /dev/xdevcfg

– Run Program: ./program <input arguments>

» Runtime Application Configuration:
  - program.xtasks.config or xtasks.config

» With instrumentation:
  - export EXTRAE_CONFIG_FILE=”extrae.xml”
  - NX_ARGS=”--instrumentation=extrae” ./program <input>

OmpSs@FPGA Magic Work

Compile Time

- Type and number of instances per fpga task can be easily indicated
- Automatically generation of an bitstream with the all the FPGA tasks, and with instrumentation support
- Generation of a Configuration File (.xtasks.config) with the bitstream information (tasks, #instances and function) for the OmpSs runtime

Runtime

- Automatic Task Parallelism and Synchronization
  - Heterogeneous or not, and using different accelerators (.xtasks.config)
- Submit FPGA Tasks and Memory Transfers
  - Transparent pinned memory allocation management for copy in/out/inouts – *User does nothing*
  - Let users to deal with memory management with Automatic memory bus management : Advanced and Optimized code

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Kernel Vivado HLS Analysis and Wrapper

Sample vector multiplication

```
#include <stdio.h>
#include <stdlib.h>
#include "vector_mult.fpga"

#pragma omp target device(fpga) copy_deps num_instances(2)
#pragma omp task in(vect_a[0:CONST_BS-1], vect_b[0:CONST_BS-1]) out(vect_c[0:CONST_BS-1])
void vector_mult(int *vect_a, int *vect_b, int *vect_c)
{
   int i, ii;
#pragma HLS ARRAY_PARTITION variable=vect_a cyclic factor=BLOCK_FACTOR
#pragma HLS ARRAY_PARTITION variable=vect_b cyclic factor=BLOCK_FACTOR
#pragma HLS ARRAY_PARTITION variable=vect_c cyclic factor=BLOCK_FACTOR
   for (i=0; i<CONST_BS; i+=BLOCK_FACTOR)
   {
   #pragma HLS PIPELINE II=1
     for (ii=0; ii<BLOCK_FACTOR; ii++)
      vect_c[i+ii]=vect_a[i+ii]*vect_b[i+ii];
   }
}


int main(int argc, char *argv[])
{

  int n=N,n_iter=10;
  int i,iter;
  int *vect_a,*vect_b, *vect_c;

  if (argc==2)
    n = atoi(argv[1]);
  if (argc==3)
    n_iter = atoi(argv[2]);

  vect_a = (int *)malloc(n*sizeof(int));
  vect_b = (int *)malloc(n*sizeof(int));
  vect_c = (int *)malloc(n*sizeof(int));

  for(i=0; i<n; i++)
    vect_a[i]=vect_b[i]=i;

  for (i=0; i<n; i+=CONST_BS)
     vector_mult(&vect_a[i], &vect_b[i], &vect_c[i]);

  #pragma omp taskwait
```

Sample vector multiplication

- Including OmpSs and Vivado HLS

```
#include <stdio.h>
#include <stdlib.h>
#include "vector_mult.fpga"

#pragma omp target device(fpga) copy_deps num_instances(2)
#pragma omp task in(vect_a[0:CONST_BS-1], vect_b[0:CONST_BS-1]) out(vect_c[0:CONST_BS-1])
void vector_mult(int *vect_a, int *vect_b, int *vect_c)
{
   int i, ii;
#pragma HLS ARRAY_PARTITION variable=vect_a cyclic factor=BLOCK_FACTOR
#pragma HLS ARRAY_PARTITION variable=vect_b cyclic factor=BLOCK_FACTOR
#pragma HLS ARRAY_PARTITION variable=vect_c cyclic factor=BLOCK_FACTOR
   for (i=0; i<CONST_BS; i+=BLOCK_FACTOR)
   {
   #pragma HLS PIPELINE II=1
     for (ii=0; ii<BLOCK_FACTOR; ii++)
      vect_c[i+ii]=vect_a[i+ii]*vect_b[i+ii];
   }
}


int main(int argc, char *argv[])
{

  int n=N,n_iter=10;
  int i,iter;
  int *vect_a,*vect_b, *vect_c;

  if (argc==2)
    n = atoi(argv[1]);
  if (argc==3)
    n_iter = atoi(argv[2]);

  vect_a = (int *)malloc(n*sizeof(int));
  vect_b = (int *)malloc(n*sizeof(int));
  vect_c = (int *)malloc(n*sizeof(int));

  for(i=0; i<n; i++)
    vect_a[i]=vect_b[i]=i;

  for (i=0; i<n; i+=CONST_BS)
     vector_mult(&vect_a[i], &vect_b[i], &vect_c[i]);

  #pragma omp taskwait
```

target the FPGA for this task and generate 2 IP cores

access vect_a, _b, and _c with in/out directionality

Data will be copied from host memory to FPGA Block-RAM

Partition vectors in the Block-RAM to allow parallel accesses

Pipeline the operations, in such a way, that we accomplish one multiplication per cycle

FPGA IP core will be invoked at every task call site

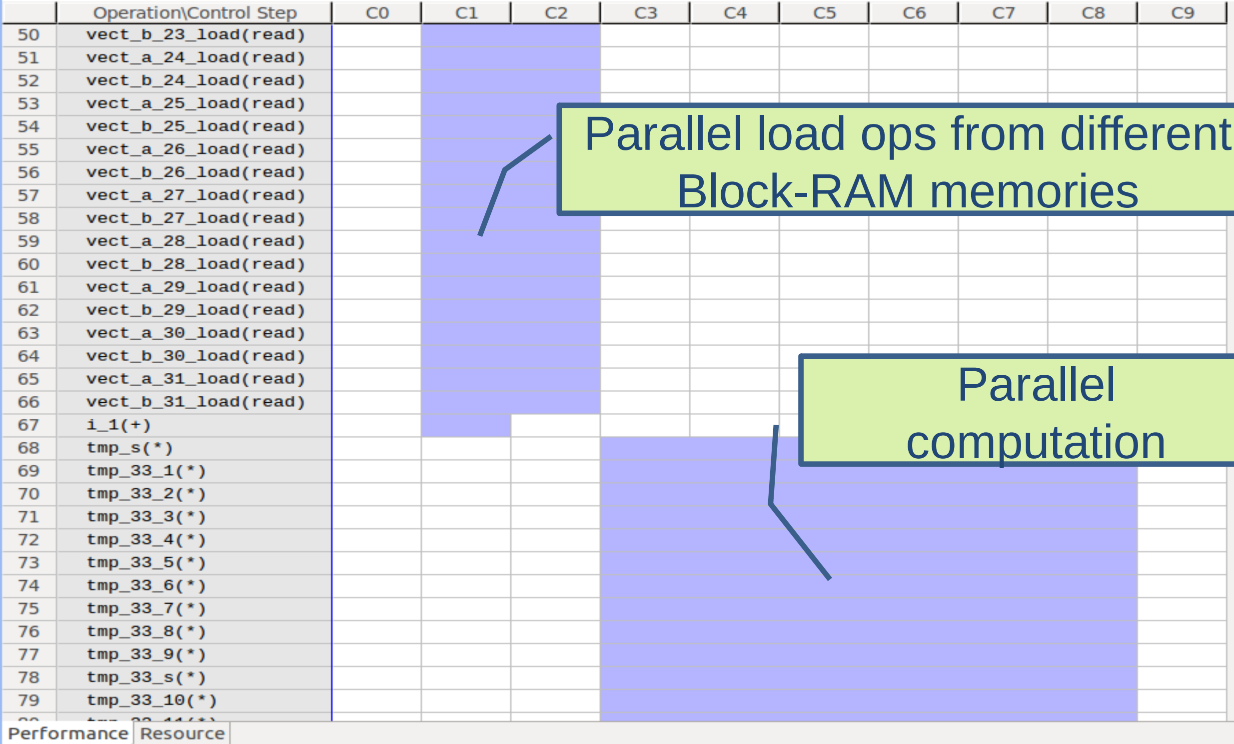
Vivado HLS Report Analysis

	BRAM	DSP	FF
vector_mult_hls_automatic_mcxx_wrapper	112	128	630
vector_mult	0	128	310

Iterations are parallelized

Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration
vector_mult	-	73	73	-
Loop 1	yes	71	1	9



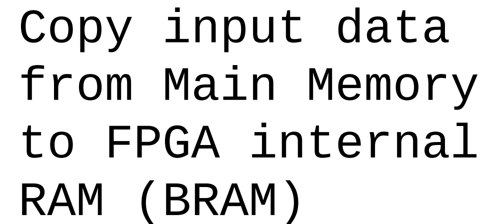
Automatic Wrapper Interface

```
void vector_mult_hls_automatic_mcxx_wrapper ( hls::stream<axiData> &inStream,
                                               hls::stream<axiData> &outStream,
                                               counter_t *mcxx_data ,
                                               int *mcxx_vect_a,int *mcxx_vect_b,int *mcxx_vect_c){
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=inStream
#pragma HLS interface axis port=outStream
#pragma HLS INTERFACE m_axi port=mcxx_data
#pragma HLS INTERFACE m_axi port=mcxx_vect_a
#pragma HLS INTERFACE m_axi port=mcxx_vect_b
#pragma HLS INTERFACE m_axi port=mcxx_vect_c
int  vect_a[2048];

int  vect_b[2048];

int  vect_c[2048];
```

- *Top wrapper autogenerated*
  - *Stream in: Argument information*
  - *Stream out: Finish signal*
  - *Instrumentation Port: data*
  - *Argument buses*

Automatic Wrapper: Copies

```
memcpy(vect_a, (const int *)(mcxx_vect_a+__laddr/sizeof(int )), (2048) * sizeof(int ) );

memcpy(vect_b, (const int *)(mcxx_vect_b+__laddr/sizeof(int )), (2048) * sizeof(int ) );
```




```
vector_mult(vect_a, vect_b, vect_c);
```

Copy output data from
FPGA internal RAM
(BRAM) to Main Memory

Automatic Wrapper: Multidimensional Example

```
 void example_mat_hls_automatic_mcxx_wrapper(hls::stream<axiData> &inStream, hls::stream<axiData>
&outStream, counter_t *mcxx_data, uint32_t accID, int *mcxx_A, int *mcxx_B) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=inStream
#pragma HLS INTERFACE axis port=outStream
#pragma HLS INTERFACE m_axi port=mcxx_data
#pragma HLS INTERFACE m_axi port=mcxx_A
#pragma HLS INTERFACE m_axi port=mcxx_B

        int  A[32][16][64];
        int  B[32][16][64];

….
                    memcpy(A, (const int *)(mcxx_A + __addr/sizeof(int )), (32)*(16)*(64)*sizeof(int ));
….
```

www.bsc.es

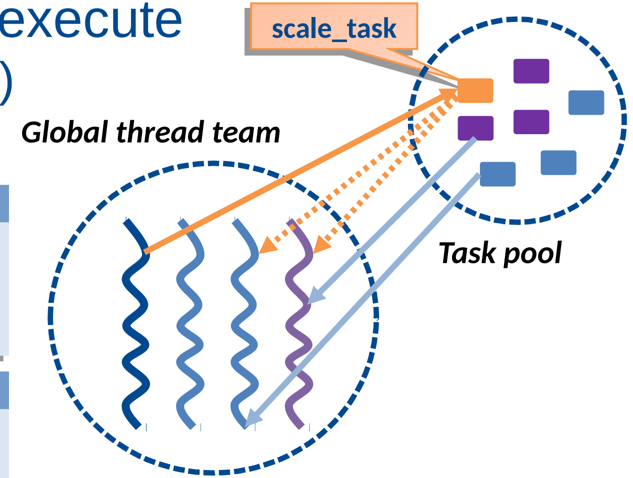
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Advanced OmpSs and OmpSs@FPGA

Implements & versioning: Device Tuning

Example: One single task & two different implementations

- Scheduler decides (at runtime) which version to execute
  - » On resource availability (first thread requesting work)
  - » Smart scheduling (shortest execution time)



scale.h

```
#pragma omp target device(smp) no_copy_deps implements(scale_task)
#pragma omp task in([SIZE]c,a) out([SIZE] b)
void scale_task_smp(double *b, double *c, double *a);
```

scale.c

```
void scale_task_smp(double *b, double *c, double *a){
   double alpha=*a;
   for (int j=0; j < SIZE; j++) b[j] = alpha*c[j];}
```

scale.fpga.h

```
#pragma omp target device(fpga) copy_deps num_instances(1)
#pragma omp task in([SIZE]c,a) out([SIZE] b)
void scale_task(double *b, double *c, double *a);
```

scale.c

```
void scale_task(double *b, double *c, double *a) {
#pragma HLS ARRAY_PARTITION variable=c complete dim=1
#pragma HLS ARRAY_PARTITION variable=b complete dim=1
   double alpha=*a;
   for (int j=0; j < SIZE; j++) b[j] = alpha*c[j]; }
```

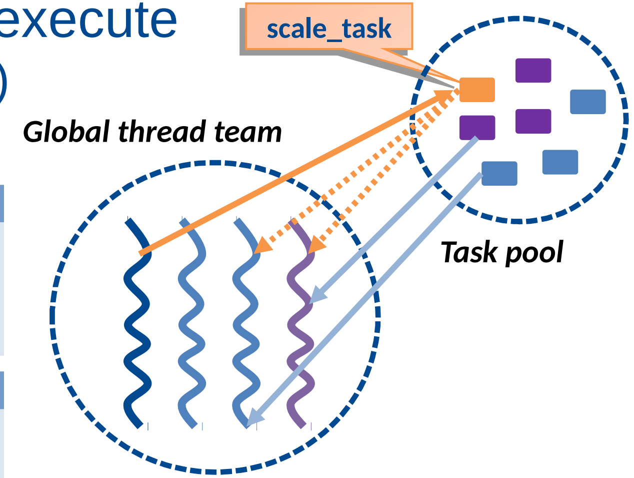
main.c

```
#include "scale.h"
#include "scale.fpga.h"

#define SIZE 100
int main (int argc, char *argv[])
{
   for (. . . ) {
     scale_task(B,C,&alpha);
   }. . .
}
```

Implements & versioning: Device Tuning

Example: One single task & two different implementations

- Scheduler decides (at runtime) which version to execute
  - » On resource availability (first thread requesting work)
  - » Smart scheduling (shortest execution time)



scale.h

```
#pragma omp target device(smp) no_copy_deps implements(scale_task)
#pragma omp task in([SIZE]c,a) out([SIZE] b)
void scale_task_smp(double *b, double *c, double *a);
```

scale.c

```
void scale_task_smp(double *b, double *c, double *a){
   double alpha=*a;
   for (int j=0; j < SIZE; j++) b[j] = alpha*c[j];}
```

scale.fpga.h

```
#pragma omp target device(fpga) copy_deps num_instances(1)
#pragma omp task in([SIZE]c,a) out([SIZE] b)
void scale_task(double *b, double *c, double *a);
```

scale.c

```
void scale_task(double *b, double *c, double *a) {
#pragma HLS ARRAY_PARTITION variable=c complete dim=1
#pragma HLS ARRAY_PARTITION variable=b complete dim=1
   double alpha=*a;
   for (int j=0; j < SIZE; j++) b[j] = alpha*c[j]; }
```

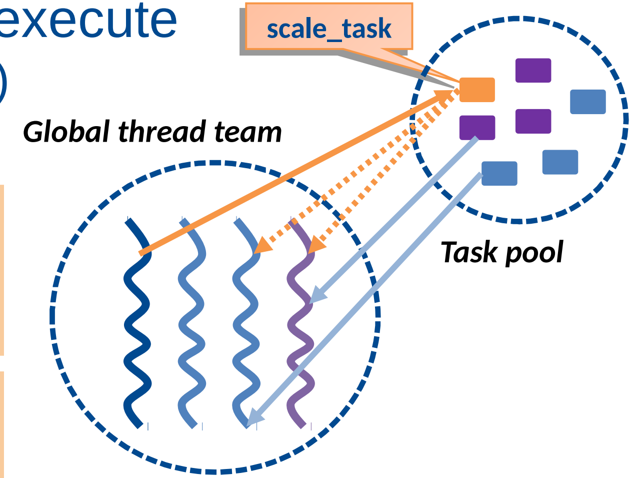
main.c

```
#include “scale.h”
#include “scale.fpga.h”

#define SIZE 100
int main (int argc, char *argv[])
{
   for (. . . ) {
     scale_task(B,C,&alpha);
   }. . .
}
```

Implements & versioning: Device Tuning

Example: One single task & two different implementations

- Scheduler decides (at runtime) which version to execute
  - » On resource availability (first thread requesting work)
  - » Smart scheduling (shortest execution time)



scale.h

```
#pragma omp target device(smp) no_copy_deps implements(scale_task)
#pragma omp task in([SIZE]c,a) out([SIZE] b)
void scale_task_smp(double *b, double *c, double *a);
```

scale.c

```
void scale_task_smp(double *b, double *c, double *a){
   double alpha=*a;
   for (int j=0; j < SIZE; j++) b[j] = alpha*c[j];}
```

scale.fpga.h

```
#pragma omp target device(fpga) copy_deps num_instances(1)
#pragma omp task in([SIZE]c,a) out([SIZE] b)
void scale_task(double *b, double *c, double *a);
```

scale.c

```
void scale_task(double *b, double *c, double *a) {
#pragma HLS ARRAY_PARTITION variable=c complete dim=1
#pragma HLS ARRAY_PARTITION variable=b complete dim=1
   double alpha=*a;
   for (int j=0; j < SIZE; j++) b[j] = alpha*c[j]; }
```

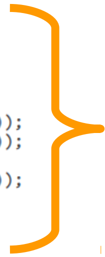
main.c

```
#include “scale.h”
#include “scale.fpga.h”

#define SIZE 100
int main (int argc, char *argv[])
{
   for (. . . ) {
     scale_task(B,C,&alpha);
   }. . .
}
```

FPGA Blocking: using memcpy

```
#include <stdio.h>
#include <stdlib.h>
#include "vector_mult_hw.fpga.h"

void vector_mult(int *vect_a, int *vect_b, int *vect_c)
{
   int i, ii;
#pragma HLS ARRAY_PARTITION variable=vect_a cyclic factor=CONST_BLOCK_FACTOR_MF
#pragma HLS ARRAY_PARTITION variable=vect_b cyclic factor=CONST_BLOCK_FACTOR_MF
#pragma HLS ARRAY_PARTITION variable=vect_c cyclic factor=CONST_BLOCK_FACTOR_MF
   for (i=0; i<CONST_BS; i+=CONST_BLOCK_FACTOR_MF)
   {
   #pragma HLS PIPELINE II=1
     for (ii=0; ii<CONST_BLOCK_FACTOR_MF; ii++)
      vect_c[i+ii]=vect_a[i+ii]*vect_b[i+ii];
   }
}

#pragma omp target device(fpga) copy_in(n[0:0]) num_instances(1)
#pragma omp task in(vect_a[0:CONST_BS-1], vect_b[0:CONST_BS-1], n[0:0]) out(vect_c[0:CONST_BS-1])
void vector_mult_multiple_fpga(int *vect_a, int *vect_b, int *vect_c, int *n)
{
   int i;
   int n_elems = n[0];
   int local_a[CONST_BS];
   int local_b[CONST_BS];
   int local_c[CONST_BS];

   for (i=0; i<n_elems; i+=CONST_BS)
   {
       memcpy(local_a, &vect_a[i], CONST_BS*sizeof(int));
       memcpy(local_b, &vect_b[i], CONST_BS*sizeof(int));
       vector_mult(local_a, local_b, local_c);
       memcpy(&vect_c[i], local_c, CONST_BS*sizeof(int));
   }
}

int main(int argc, char *argv[])
{
  int n=N,n_iter=10;
  int i,iter;
  int *vect_a,*vect_b, *vect_c;

  vect_a = nanos_fpga_alloc_dma_mem(n*sizeof(int));
  vect_b = nanos_fpga_alloc_dma_mem(n*sizeof(int));
  vect_c = nanos_fpga_alloc_dma_mem(n*sizeof(int));

  for(i=0; i<n; i++)
    vect_a[i]=vect_b[i]=i;

 for(iter=0;iter<n_iter; iter++)
  vector_mult_multiple_fpga(vect_a, vect_b, vect_c, &n);

  #pragma omp taskwait
```

Only ***n*** argument
is
Copy in



FPGA Blocking....
… data locality
optimizations, etc ...

FPGA Blocking: Mercurium FPGA phase helps

```
#include <stdio.h>
#include <stdlib.h>
#include "vector_mult_hw.fpga.h"

void vector_mult(int *vect_a, int *vect_b, int *vect_c)
{
   int i, ii;
#pragma HLS ARRAY_PARTITION variable=vect_a cyclic factor=CONST_BLOCK_FACTOR_MF
#pragma HLS ARRAY_PARTITION variable=vect_b cyclic factor=CONST_BLOCK_FACTOR_MF
#pragma HLS ARRAY_PARTITION variable=vect_c cyclic factor=CONST_BLOCK_FACTOR_MF
   for (i=0; i<CONST_BS; i+=CONST_BLOCK_FACTOR_MF)
   {
   #pragma HLS PIPELINE II=1
     for (ii=0; ii<CONST_BLOCK_FACTOR_MF; ii++)
      vect_c[i+ii]=vect_a[i+ii]*vect_b[i+ii];
   }
}

#pragma omp target device(fpga) copy_in(n[0:0]) num_instances(1)
#pragma omp task in(vect_a[0:CONST_BS-1], vect_b[0:CONST_BS-1], n[0:0]) out(vect_c[0:CONST_BS-1])
void vector_mult_multiple_fpga(int *vect_a, int *vect_b, int *vect_c, int *n)
{
   int i;
   int n_elems = n[0];
   int local_a[CONST_BS];
   int local_b[CONST_BS];
   int local_c[CONST_BS];

   for (i=0; i<n_elems; i+=CONST_BS)
   {
       memcpy(local_a, &vect_a[i], CONST_BS*sizeof(int));
       memcpy(local_b, &vect_b[i], CONST_BS*sizeof(int));
       vector_mult(local_a, local_b, local_c);
       memcpy(&vect_c[i], local_c, CONST_BS*sizeof(int));
   }
}
```

- ***Arguments may be copied or not***
- ***Arguments should be arrays (pointers)***
- ***Arguments with no copies***
  - ***Addr is automatically passed in***
  - ***Addr transparent access to SMP Memory using AXI ports***
  - ***Transparent to the Programmer!***
    - ***Be careful with memcpy !!!***

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Results: Instrumentation

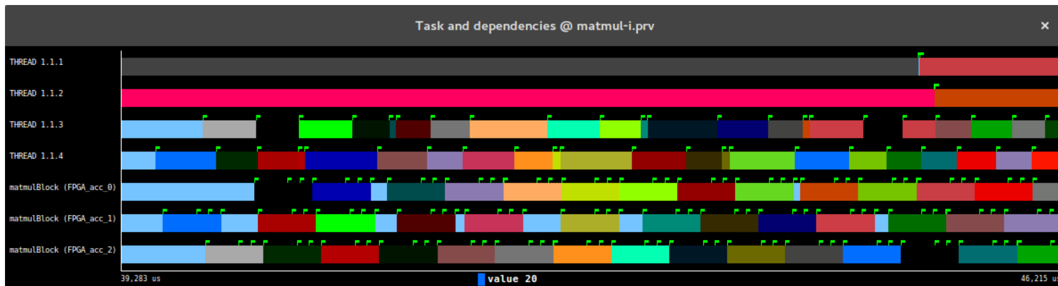
OmpSs@FPGA Trace Instrumentation

Task and dependencies @ matmul-i.prv

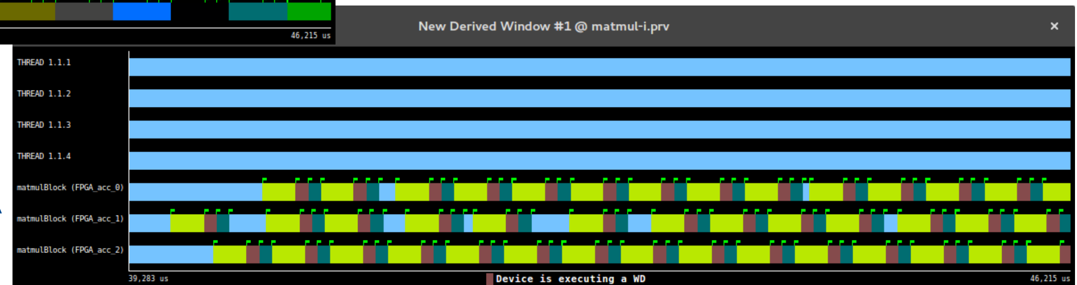
Threads
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4

FPGA
matmulBlock (FPGA_acc_0)
matmulBlock (FPGA_acc_1)
matmulBlock (FPGA_acc_2)

28,352 us
value 16
141,264 us



Tasks can be “followed” between threads and accelerators



The “insides” of the FPGA can be analyzed

High Level Overview: Hardware Instrumentation

rst_processing_system7_0_50M
Processor System Reset

mmult_core_0
Mmult_core (Pre-Production)

axi_mem_intercon
AXI Interconnect

axi_dma
AXI Direct Memory Access

axi_mem_intercon_1
AXI Interconnect

axi_bram_ctrl_0
AXI BRAM Controller

hw_timer64_0_bram
Block Memory Generator

processing_system7_0
ZYNQ7 Processing System

util_vector_logic_0
Utility Vector Logic

hw_timer64_0

Counter BRAM

Counter implementation

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Results: HPC Results

OmpSs@FPGA Evaluation: Zynq Ultrascale+

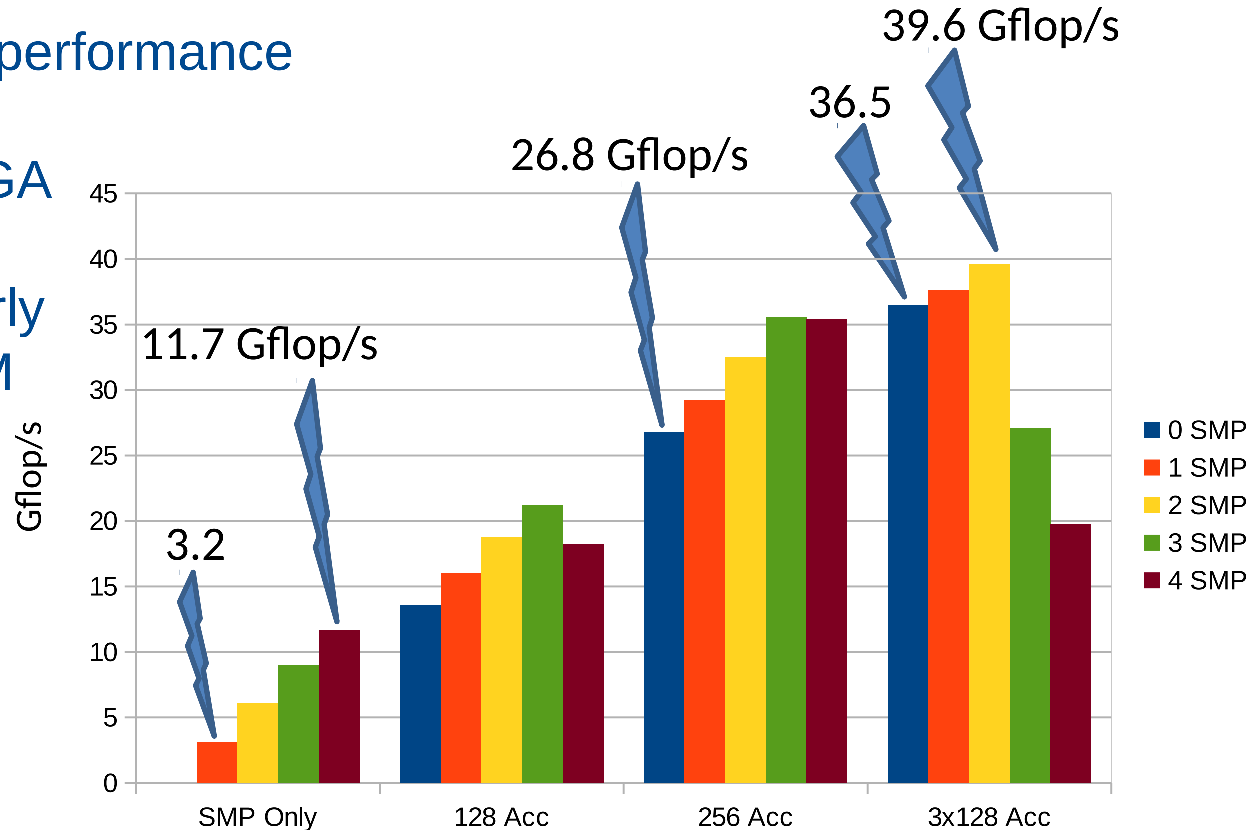
MxM 2048x2048 single precision: BS 128 (300MHz) & 256 (300Mhz)

– Threads (OpenBLAS) performance is not affected by the functionality in the FPGA

– FPGA @300MHz clearly outperforms the 4 ARM cores

– “Implements” is **greatly** successful

– The fastest version
– needs 2 cores feeding
– the accelerators



OmpSs@FPGA Evaluation: Zynq Ultrascale+

Reports from compilation are available

– Kernel 256x256, on Ultrascale+ xczu9eg

```
FF:   153654 used | 548160 available - 28.03% utilization
LUT:  111634 used | 274080 available - 40.73% utilization
BRAM:    648 used |   1824 available - 35.52% utilization
DSP:    1280 used |   2520 available - 50.79% utilization
2 kernel(s) synthesized. 144s elapsed.
Generating Vivado tcl script...
Vivado tcl script generated. 0s elapsed.
Starting Block Design generation...

****** Vivado v2016.3 (64-bit)
```

– Kernel 128x128

```
FF:    66642 used | 548160 available - 12.15% utilization
LUT:   51981 used | 274080 available - 18.96% utilization
BRAM:    305 used |   1824 available - 16.72% utilization
DSP:     640 used |   2520 available - 25.39% utilization
2 kernel(s) synthesized. 82s elapsed.
Generating Vivado tcl script...
Vivado tcl script generated. 0s elapsed.
Starting Block Design generation...
```

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

OmpSs@FPGA

Tutorial – Introduction

BSC OmpSs@FPGA team

Universitat Politècnica de Catalunya, and Barcelona Supercomputing Center

November 4th, 2018