www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

OmpSs@FPGA

Deusto – Introduction

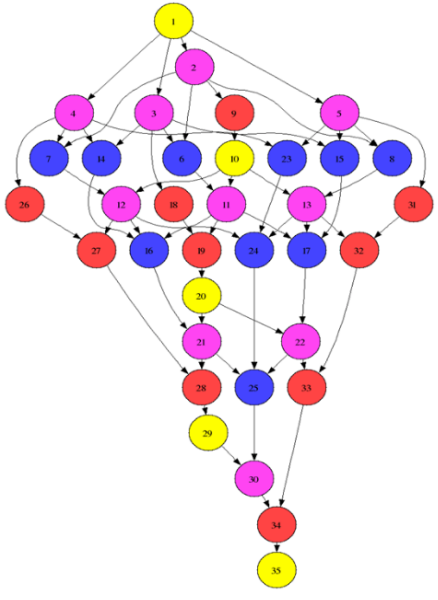
BSC OmpSs@FPGA team

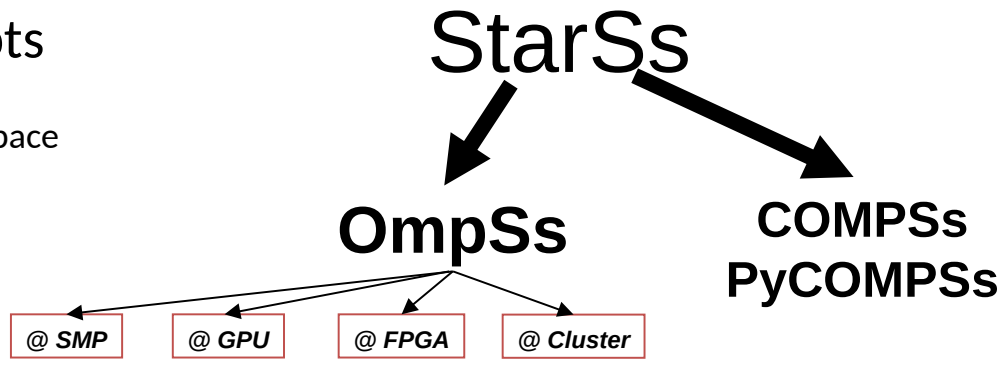
Universitat Politècnica de Catalunya, and
Barcelona Supercomputing Center

December, 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



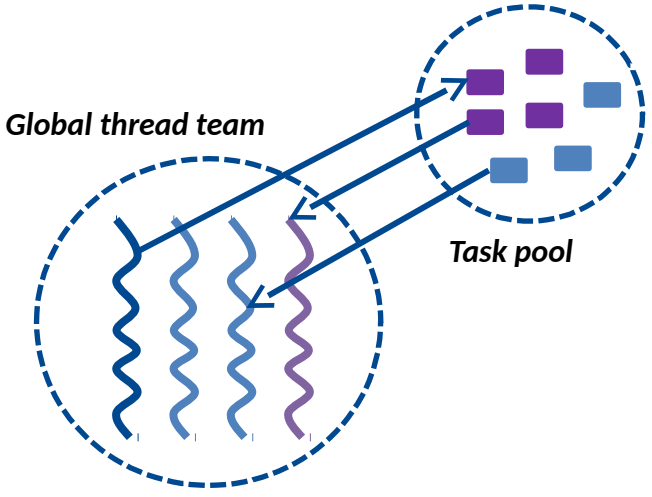


- OmpSs is used for prototyping extensions to OpenMP
  - Tasking
  - data-dependences
  - Heterogeneity
  - Multiple address spaces
  - …
- Mercurium compiler
- Nanos runtime system

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)

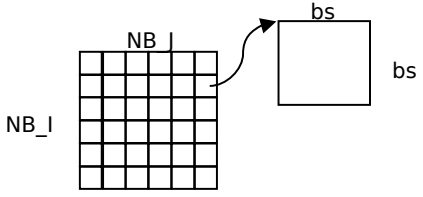
www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Easy to Program

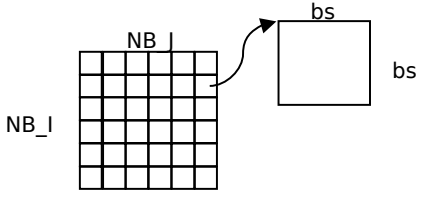
OmpSs Example: Tiled MxM

```
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
```

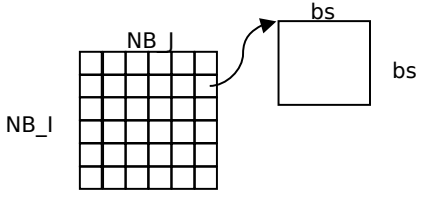


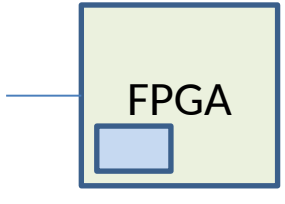
OmpSs Example: Tiled MxM

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
```





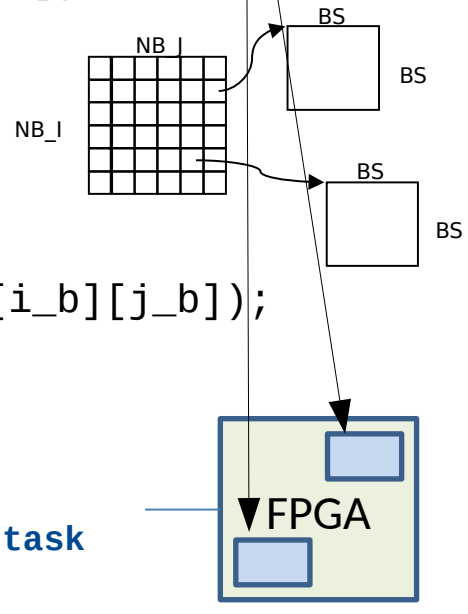
OmpSs Example: Two MxM accelerators

Mercurium compiler **with autoVivado tool**

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

```
#pragma omp target device(fpga) copy_deps
#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
```

num_instances(2)



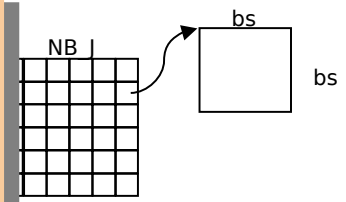
OmpSs Example: MxM kernel + Vivado HLS

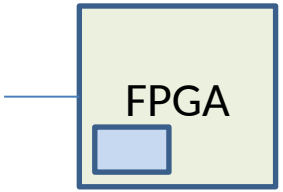
Mercurium compiler **with autoVivado tool**

- 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;
    }}
```



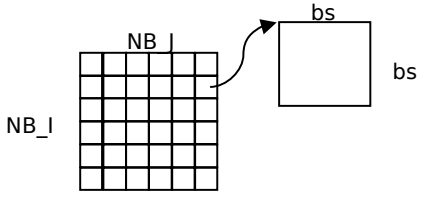


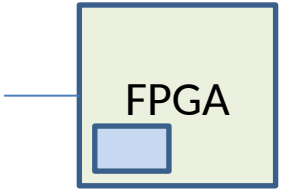
OmpSs Example: MxM kernel + Vivado HLS

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 ++

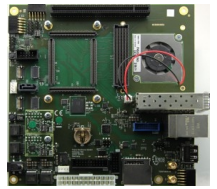
OmpSs FPGA Support (present and near future)

```
#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;
    }}
```

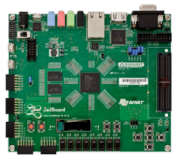
Discrete FPGAs and Cloud

ZCU102 Board
XCZU9EG-2FFVB1156

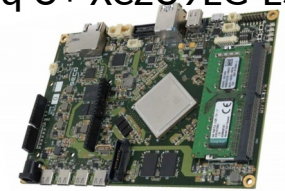
Trenz Electronics Zynq U+
TE0808 XCZU9EG-ES1



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)

www.bsc.es

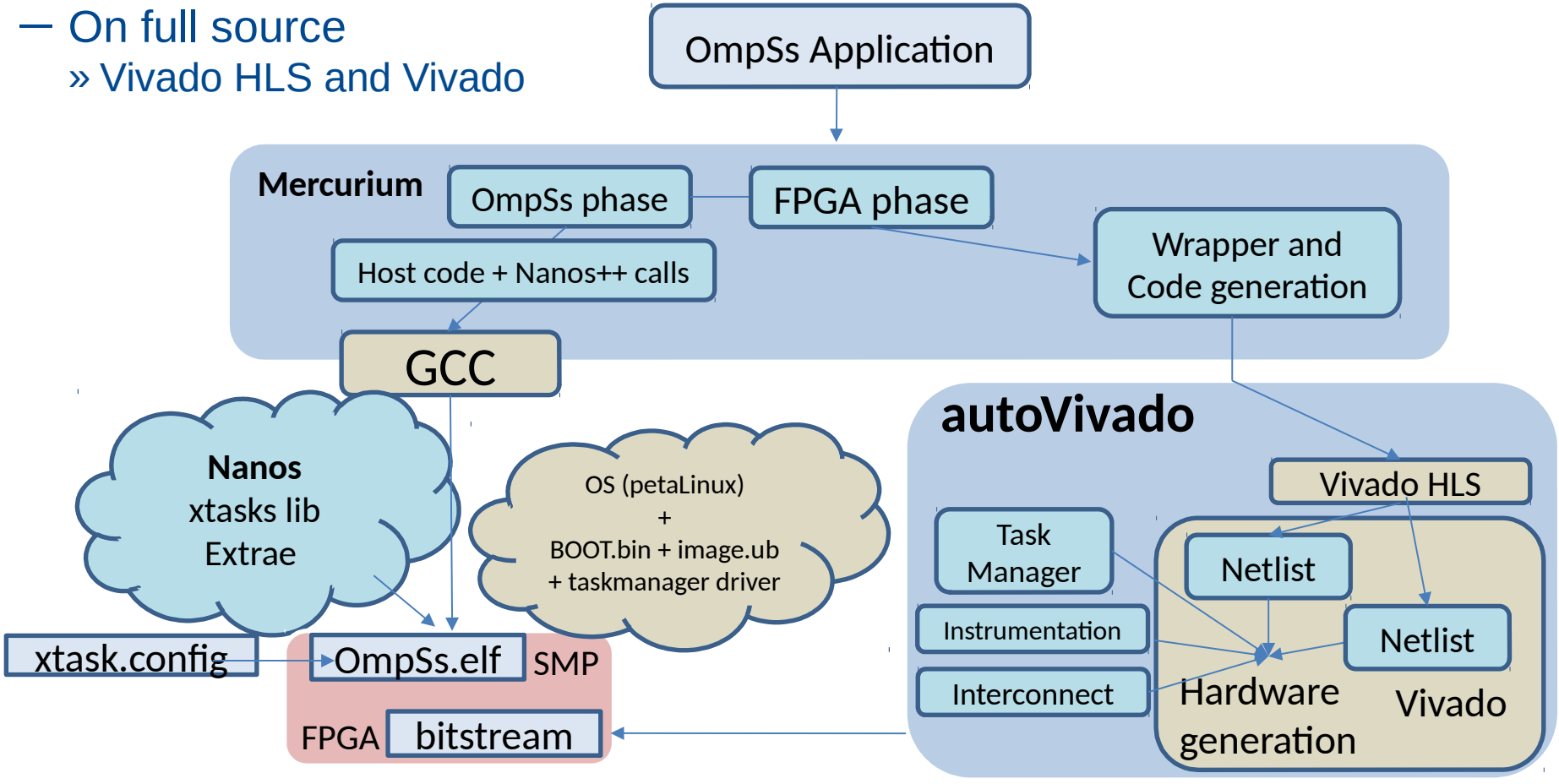
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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:

    --bitstream-generation
  - » Linking options:

    --Wf,"--board=$(BOARD_NAME),--clock=100,--task_manager"

    --Wf,"-v,--name=vivado_project_name,--dir=$(VIVADO_WORKSPACE)"

Execution

Few details on the steps to run the application

- Environment:
  - 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>

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Kernel Vivado HLS Analysis

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 HL

```
#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

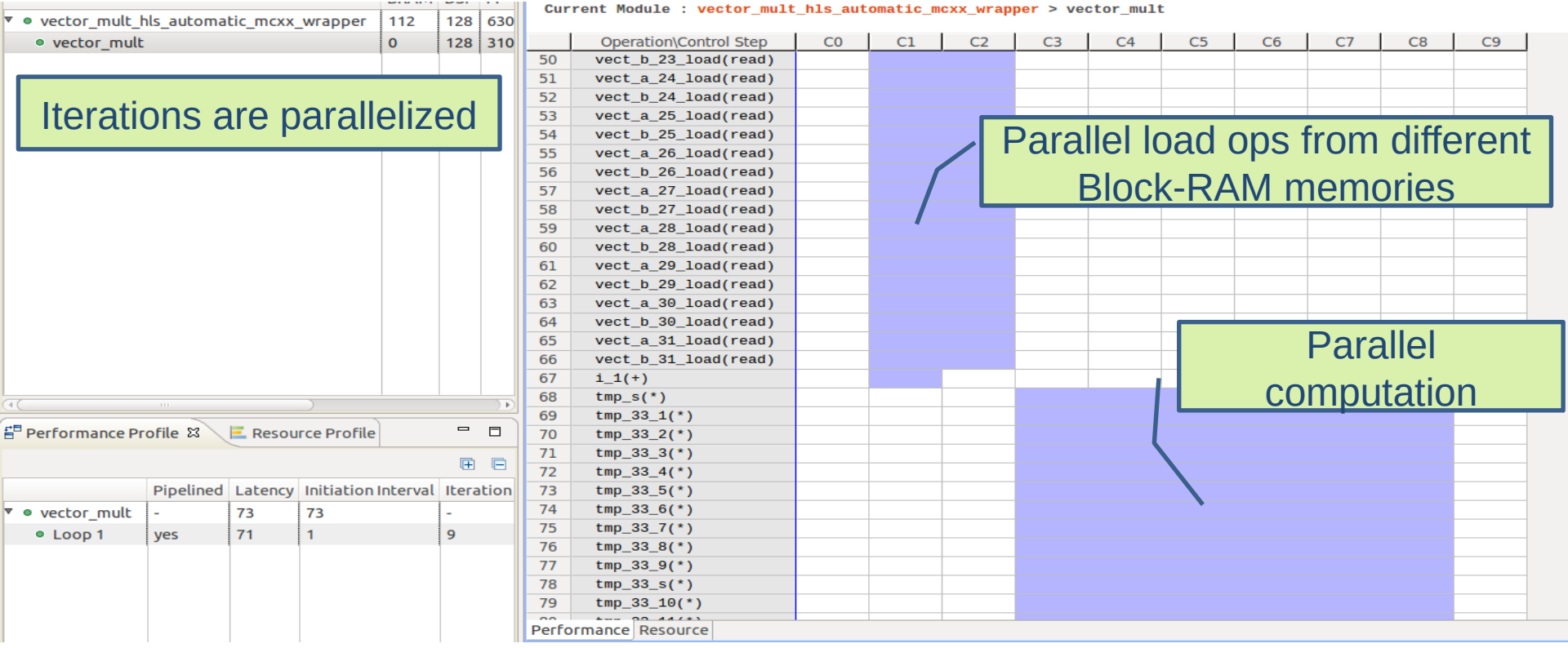
Data will be copied from host memory to FPGA Block-RAM

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



www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MxM Dataflow and Results

How to improve the HLS kernel

Group data transfers

- Allow HLS to better pipeline the code
- **Dataflow** across transfers and computation
- At the expense of more complex code for the programmer

```
#pragma omp target device(fpga) copy_deps no_localmem_copies num_instances(3)
#pragma omp task in([BSIZE*BSIZE]a, [BSIZE*BSIZE]b) inout([BSIZE*BSIZE]c)
void matmulBlock(elem_t *a, elem_t *b, elem_t *c) {
   unsigned int i, j, k, l;
#pragma HLS dataflow
   elem_t aa[BSIZE][BSIZE];
   elem_t bb[BSIZE][BSIZE];
   elem_t cc[BSIZE][BSIZE];
   elem_t r0[BSIZE];
   elem_t r1[BSIZE/2];
   elem_t r2[BSIZE/4];
   elem_t r3[BSIZE/8];
   elem_t r4[BSIZE/16];
   elem_t r5[BSIZE/32];
   elem_t r6[BSIZE/64];
   elem_t r7[BSIZE/128];
```

How to improve the HLS kernel

```
#pragma omp target device(fpga) copy_deps no_localmem_copies num_instances(3)
#pragma omp task in([BSIZE*BSIZE]a, [BSIZE*BSIZE]b) inout([BSIZE*BSIZE]c)
void matmulBlock(elem_t *a, elem_t *b, elem_t *c) {
   unsigned int i, j, k, l;
#pragma HLS dataflow
   elem_t aa[BSIZE][BSIZE];
   elem_t bb[BSIZE][BSIZE];
   elem_t cc[BSIZE][BSIZE];
   elem_t r0[BSIZE];
   elem_t r1[BSIZE/2];
   elem_t r2[BSIZE/4];
   elem_t r3[BSIZE/8];
   elem_t r4[BSIZE/16];
   elem_t r5[BSIZE/32];
   elem_t r6[BSIZE/64];
   elem_t r7[BSIZE/128];
#pragma HLS array_partition variable=aa block factor=BSIZE/4 dim=2
#pragma HLS array_partition variable=bb block factor=BSIZE/4 dim=2
#pragma HLS array_partition variable=r0 block factor=BSIZE/4
#pragma HLS array_partition variable=r1 block factor=BSIZE/8
#pragma HLS array_partition variable=r2 block factor=BSIZE/16
#pragma HLS array_partition variable=r3 block factor=BSIZE/32
...
```

How to improve the HLS kernel

Do pipelined transfers

```
// inputs
for (i = 0; i < BSIZE; i++) {
     #pragma HLS pipeline II=BSIZE
     memcpy(&aa[i][0], &a[i*BSIZE], BSIZE*sizeof(elem_t));
     memcpy(&bb[i][0], &b[i*BSIZE], BSIZE*sizeof(elem_t));
     memcpy(&cc[i][0], &c[i*BSIZE], BSIZE*sizeof(elem_t));
}

// computation (on next slide)
....
// output
for (i = 0; i < BSIZE; i++) {
     #pragma HLS pipeline II=BSIZE
     memcpy(&c[i*BSIZE], &cc[i][0], BSIZE*sizeof(elem_t));
}
```

How to improve the HLS kernel

Computation split in stages : Assume B Transposed

```
 for (i = 0; i < BSIZE; i++) {
      for (j = 0; j < BSIZE; j++) {    // 2 cycles to start the next loop j
#pragma HLS pipeline II=2              // 2 for 256x256    1 for 128x128
    // Parallel computation            // 4 for Zybo (due to resources)
         for (k = 0; k < BSIZE/2; k++) {
            r0[k] = aa[i][k] * bb[j][k];
            r0[k+BSIZE/2] = aa[i][k+BSIZE/2] * bb[j][k+BSIZE/2];
         }

         // Reduction
         for (k = 0; k < BSIZE/2;   k++) r1[k] = r0[k] + r0[BSIZE/2+k];
         for (k = 0; k < BSIZE/4;   k++) r2[k] = r1[k] + r1[BSIZE/4+k];
         for (k = 0; k < BSIZE/8;   k++) r3[k] = r2[k] + r2[BSIZE/8+k];
         for (k = 0; k < BSIZE/16;  k++) r4[k] = r3[k] + r3[BSIZE/16+k];
         for (k = 0; k < BSIZE/32;  k++) r5[k] = r4[k] + r4[BSIZE/32+k];
         for (k = 0; k < BSIZE/64;  k++) r6[k] = r5[k] + r5[BSIZE/64+k];
         for (k = 0; k < BSIZE/128; k++) r7[k] = r6[k] + r6[BSIZE/128+k];
         cc[i][j] += r7[0] + r7[1];
      }
 }
```

Performance on the ZU+

AXIOM board, 64 bits 4xCortex-A53, ZU+ FPGA

Matrix Multiply 2048x2048 single precision

Implements

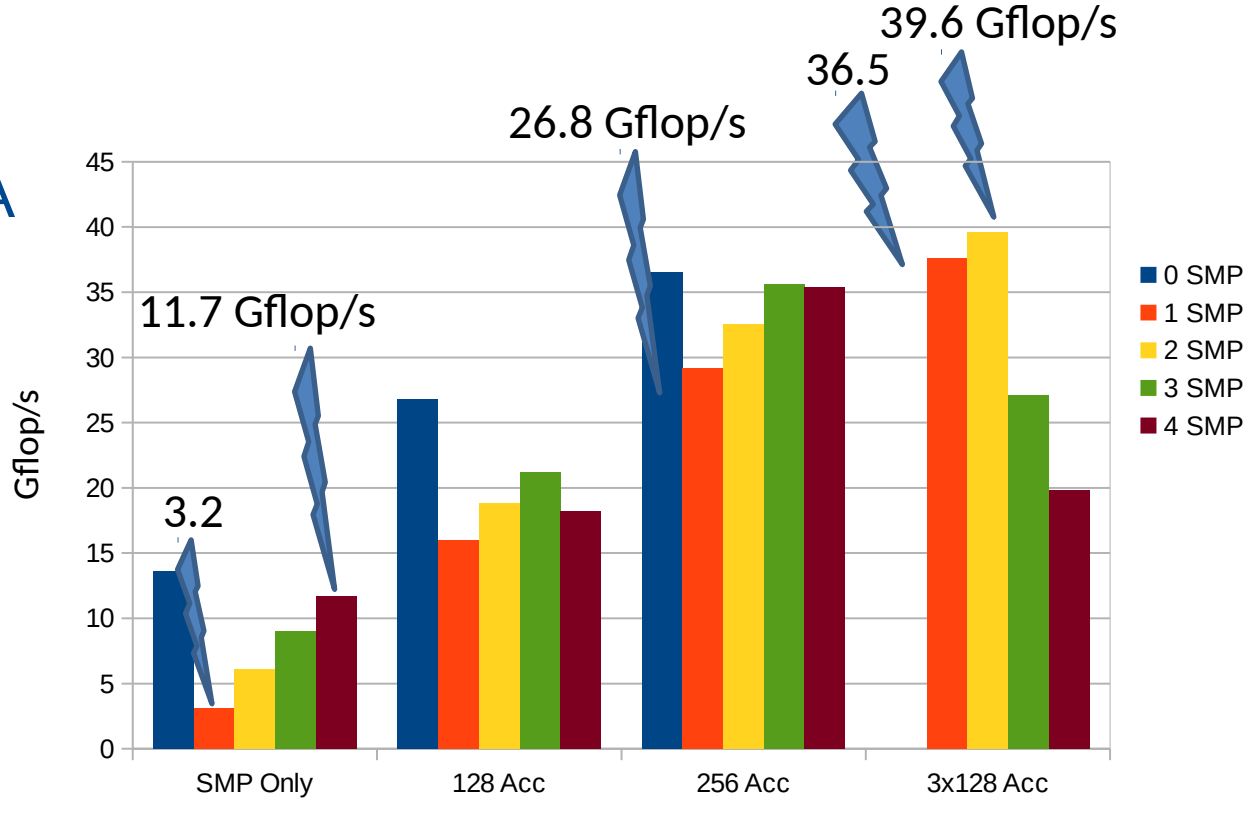
- SMP OpenBLAS
- Great success

300 Mhz on FPGA

- Outperforms cores

Parallel creation

New kernel

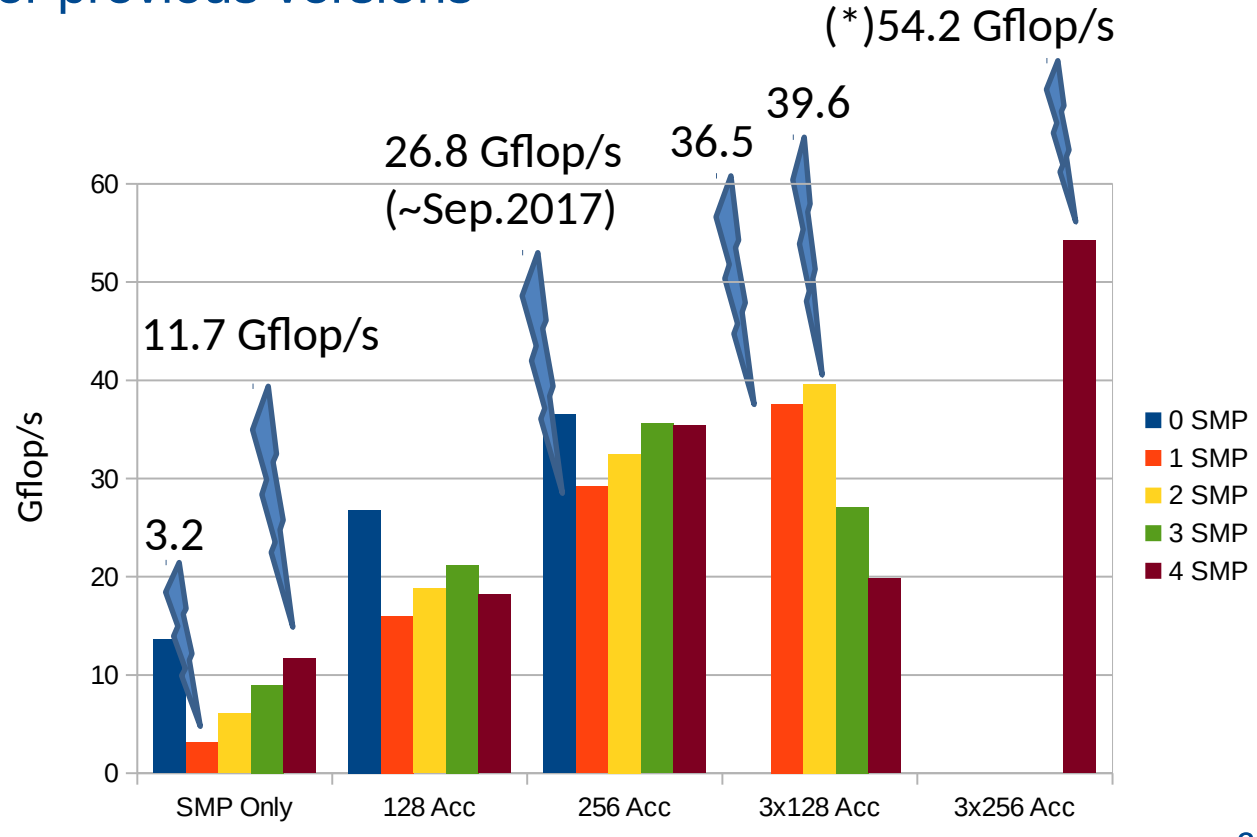


Performance on the ZU+

AXIOM board, 64 bits 4xCortex-A53, ZU+ FPGA

New kernel (*)

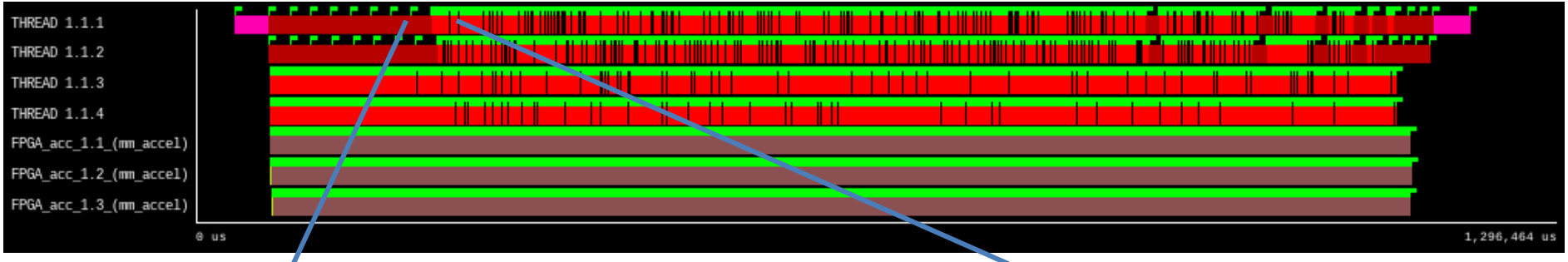
– Great improvement over previous versions

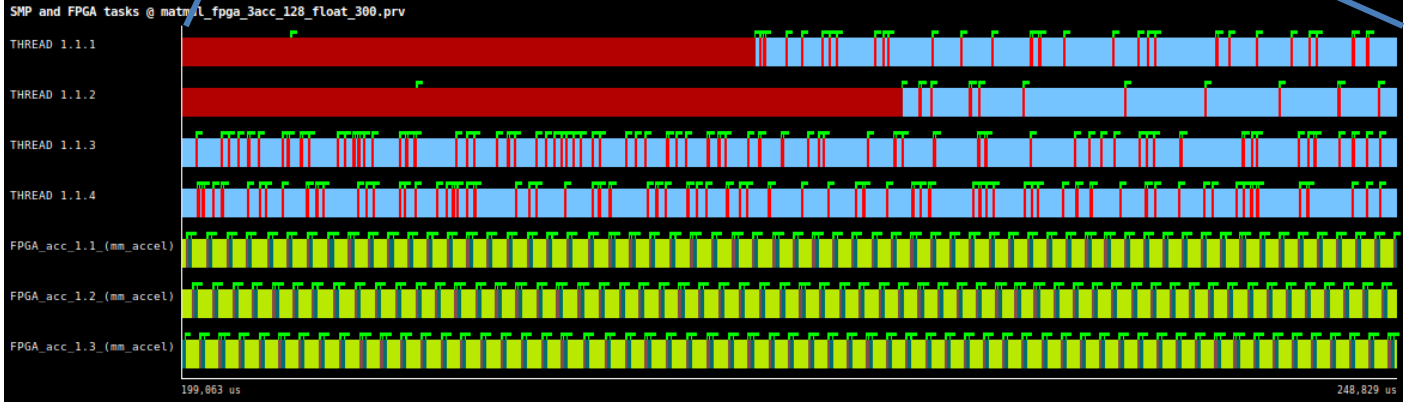


Instrumentation on the ZU+

3x128, FPGA only (no implements)

– SMP cores nearly empty

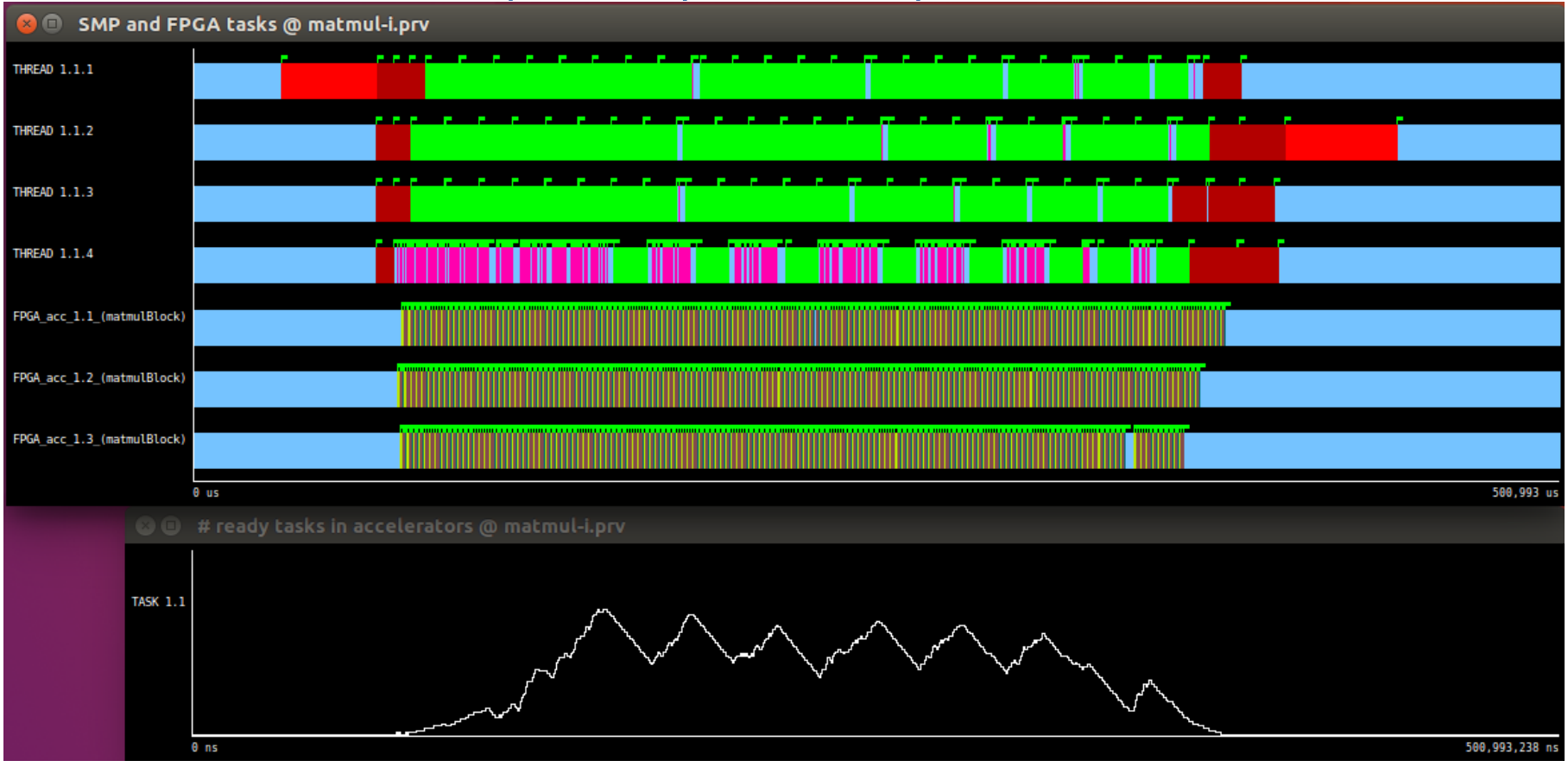




Instrumentation on the ZU+

3x256 with implements (4 workers)

– Cores do a lot of work (on their possibilities)



www.bsc.es

Vivado HLS and OmpSs@FPGA Experience

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Cross-Correlation

Code, Analysis and Experience

1) OmpSs@FPGA Porting

- Make a kernel to be accelerated by our ecosystem

2) #pragma HLS pipeline II (target vs achieved)

- Achieve a II<iteration latency, analysis of the resource conflicts using vivado_hls gui and output of the compilation

3) Strip-mining to achieve an equivalent II=1 or less than that

- Using strip-mining we may achieve an equivalent II=1 or less, if enough resources are available.
- That may imply to use more memory resources like more accumulators to do everything in parallel and not having a accumulator dependency (and its latency)
- Select operations (if) can help to reduce the II

Code, Analysis and Experience

4) #pragma HLS unroll
#pragma HLS partitioning cyclic/block/complete factor=XX dim=X

- Reduce the latecy iterations by unrolling loops that all its iterations can be done in parallel
- Analysis shows that there are conflicts accessing to memory. Partitioning can help

5) Other optimizations vs hardware resources

- Software optimizations may not be possible to be applied due to hardware resources, but they would be good for performance.
- Potential software un-optimized code can help to reduce the II (case of the innermost loop and the if conditions)

Code, Analysis and Experience

6) Vivado HLS may try to do some optimizations that will provoke resource conflicts

For instance: cyclic partitioning provokes several loads, that joint with the unroll, will provoke resource conflicts.

7) Hardware Instrumentation

By default: copies and computation time of the kernel

User instrumentation: limited number of events

Both instrumentations provide deep analysis

www.bsc.es

Matrix Multiply

Code, Analysis and Experience

1) OmpSs@FPGA Porting

- This is just to try to accelerate the MxM

2) #pragma HLS pipeline and partitioning

- As before, those two directives allow to reduce the overall latency of the loops

3) Open parallelism at SMP side may be useful to feed the accelerators very fast.

- In the case of the Zynq7000 family, with only 2 smp cores doesn't help

4) Dataflow and memory transfer by user program code

- User program introduce the memory copies in the program to access host memory
- Dataflow pragma can be used with functions and loops

Code, Analysis and Experience

5) Blocking from the FPGA

- Mechanism to reduce overhead of creating task to be spawn to the hardware accelerators
- User requires to have memory access control

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Other Experiences

Other hints

1) Communication vs Computation

Increase Frequency

Increase the number of parallel accellerators doing the same

Blocking

2) Resources

#pragma HLS resource variable=var core=XXXXX

LUTS: ram_2p_lutram

BRAMS: ram_2p_bram

DSPs: FaddSub_fulldsp

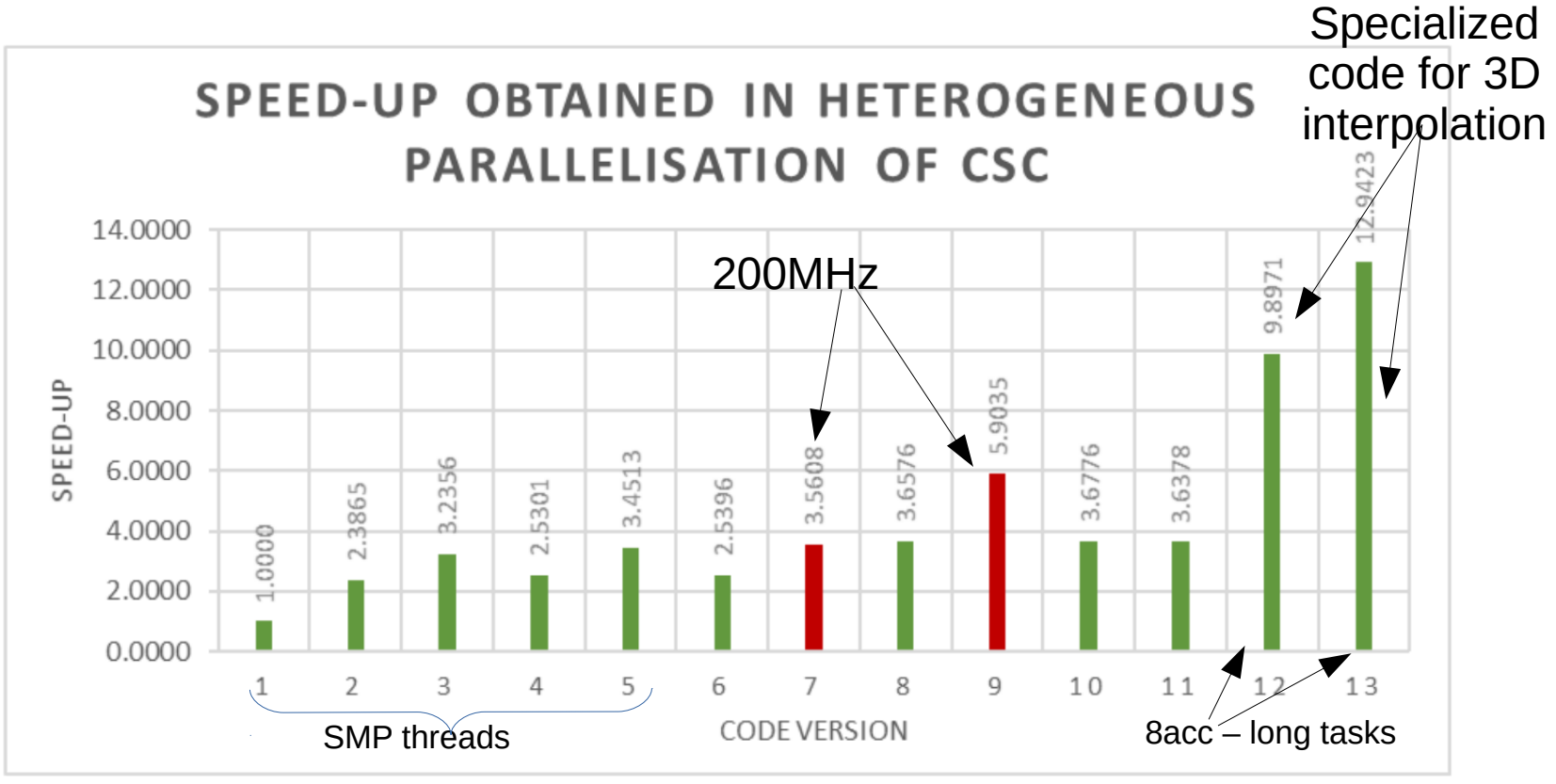
etc.

3)Dependences

#pragma HLS dependence variable=var inter/intra false

Trace of several IPs: Colour Space Converter

Zedboard Results



www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

OmpSs@FPGA

Deusto – Introduction

BSC OmpSs@FPGA team

Universitat Politècnica de Catalunya, and
Barcelona Supercomputing Center

December, 2018

1) 2 threads SMP, 2) 2acc@100MHz and 2 threads, 3) =2 @ 200Mhz, 4) 3 acc @ 100 + 2 threads 5) 3 acc @ 200 + 2 threads, 6) 3 acc @ 100, 7) 3 acc @ 200, 8) 8 acc @ 100 (block partitioning + dataflow), 9) =8 @ 200, 10) 10 acc @ 100, 11) 8 acc @ 100 no dataflow, 12) 8 acc- Optimized tunned code for 3D interpolation), 2 tasks per acc. 13) = 12 but only one long task per accelerator.