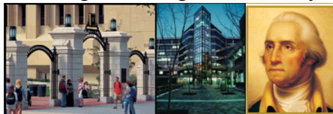
Parallel Programming of High-Performance Reconfigurable Computing Systems with UPC

Tarek El-Ghazawi, Olivier Serres, Samy Bahra, Miaoqing Huang and Esam El-Araby

High Performance Computing Laboratory
The George Washington University



July 8, 2008

Introduction
- Background and motivations
- Approach

Unified Parallel C
- Overview, memory and execution model
- Data and work distribution

Proposed solution
- Library approach
- Compilation Approach
- Code example

Experimental Testbed and Results
- Testbed
- Library approach
- Compilation approach

Conclusions and Future Work

Background

- High-Performance Reconfigurable Computers (HPRCs) integrate microprocessors and FPGAs into scalable systems
- Orders of magnitude improvement in speed, power and cost in some applications (e.g. bio-informatics, image processing and cryptography)

Motivations

- Productivity is still an issue, due to HPRCs' programming complexity
- Existent programming models are incomplete
  - FPGA High-Level Languages (HLLs) do not address system/multi-node wide High-Performance Computers (HPCs)
  - HPC Parallel Languages are not FPGA-aware

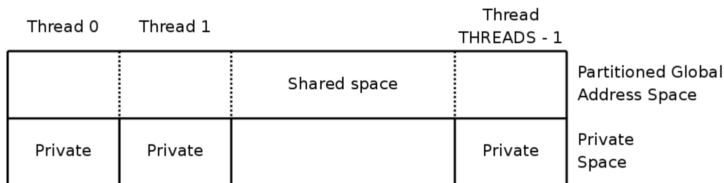
Approach

- Bringing both worlds together in a unified programming model
- Starting from a parallel language (e.g. Unified Parallel C - UPC)
- Adding FPGA awareness to the parallel language through a
  - library approach (e.g. GWU-GMU core lib)
  - compilation approach (e.g. integrating a hardware compiler, such as Impulse C)

Overview

- Parallel extension of ISO C (any C program is a valid UPC program)
- PGAS model (Partitioned Global Address Space)
- Mature and widely used (Compilers from Cray, HP, Berkeley, Intrepid, ...)

Memory and execution model



Data and work distribution

- shared [32] int x[256];
- upc_forall(init, cond, incr, affinity)
  - init, cond, incr: as in for
  - affinity: an integer or a pointer

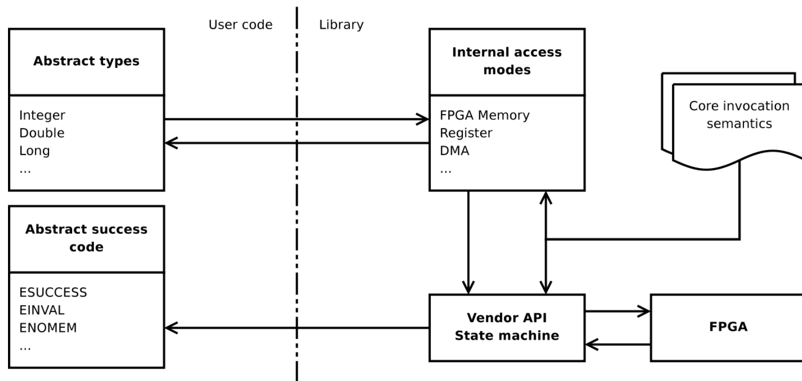
Other features

- Two memory models: relaxed or strict
- Rich synchronization mechanisms: locks, barriers, two-phase barriers, ...
- Libraries: collectives, IO, ...

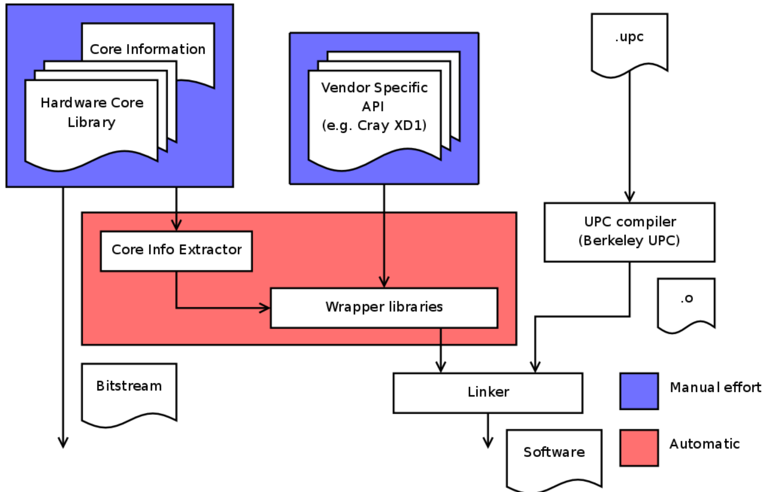
Library approach

- Allows the usage of optimized pre-designed cores
- Via a well defined interface:
  - No need to be an expert to use a core
  - No need to know the vendor API
  - Abstract and portable
- Asynchronous calls

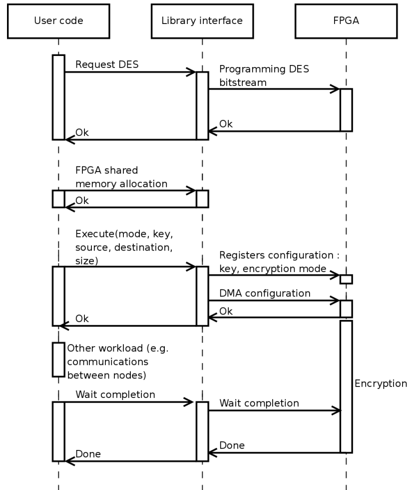
Library approach



Library approach flow



Usage example: calling a DES core



Compilation Approach

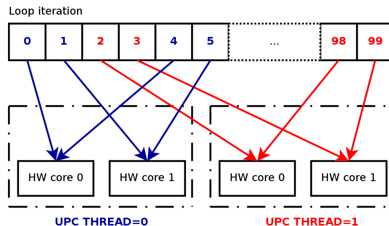
- The user may not have an optimized library
- Parallelism inherently built into the language
- Leveraging source-to-source compilers
  - Code sections destined to hardware are branched to a C-to-hardware compiler (e.g. Impulse C compiler)
  - FPGA $\leftrightarrow$ CPU communication code is also added
- Can leverage some standard UPC compiler optimisation

Extending the upc_forall semantic

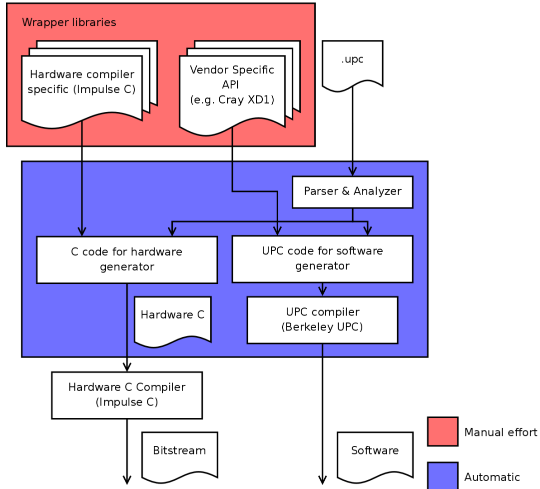
- Nesting of upc_forall represents multi-level parallelism
- Will still be valid UPC programs.

Work distribution

- upc_forall(int i=0; i < 100; i++; i/2)



Compilation approach flow



Proof of concept

- Berkeley UPC
- Impulse C
- A Java program associated with ANTLR
  - Parse the UPC code and identify the code sections destined to the hardware (function level)
  - Extract them and convert them to Impulse C code (type conversion, communication code, ...)
  - Generate the UPC code effectively calling the hardware core (FPGA programming and communication)

Compilation approach - Code example DWT 1

```c
#include <upc.h>

#include "dwt.h"
#include "image.h"

#define SIZE_X 1600
#define SIZE_Y 1200

// Shared array for source and result
shared [SIZE_X] unsigned char
    img_source[SIZE_X][SIZE_Y];
shared [SIZE_X] unsigned char
    img_result[SIZE_X][SIZE_Y];

void clearDWT();

void clearDWT(){
    // DWT has a buffer of 4 pixels, clear it
    for(int i=0; i<4; i++)
        stepDWT(0);
}
```

```c
int main(int argc, char *argv[]){
    // Read img_source
    readImage();

    // Distribute rows among the nodes
    upc_forall(int y=0; y<SIZE_Y; y++; y){
        // Empty DWT buffer for this new row
        clearDWT();

        for(int x=0; x<SIZE_X; x++){
            // Get the result for one pixel
            int result = stepDWT( img_source[x][y] );

            // Downsample by two
            if( x % 2 == 1 ){
                // Store the result in img_result
                img_result[x / 2][y] = result & 0xff;
                img_result[x / 2 + SIZE_X / 2][y] =
                    result >> 16;
            }
        }
    }

    writeImage();

    return 0;
}
```

Compilation approach - Code example DWT 2

```
#define TAPS 4
#define MAX_PIXEL 255
#pragma RUPC HARDWARE
unsigned long int stepDWT(unsigned long int nSample){
    // Declare coefficients and needed buffers
    static int LP_coef[] = {31651, 54822, 14689, -8481};
    static int HP_coef[] = {-8481, -14689, 54822, -31651};
    static int buffer[TAPS];
    int LP_accum = 0, HP_accum = 0, tap;

    // Put the input pixel in the buffer
    buffer[TAPS - 1] = nSample;

    // Compute DWT
    for(tap=0; tap < TAPS; tap++){
        LP_accum += buffer[tap] * LP_coef[tap];
        HP_accum += buffer[tap] * HP_coef[tap];
    }

    // Shift buffers
    for(tap=1; tap < TAPS; tap++)
       buffer[tap - 1] = buffer[tap];

    // Downscale
    HP_accum >>= 16;
    LP_accum >>= 16;

    // Check for saturation
    if( HP_accum < 0 ) HP_accum = 0;
    if( HP_accum > MAX_PIXEL ) HP_accum = MAX_PIXEL;
    if( LP_accum < 0 ) LP_accum = 0;
    if( LP_accum > MAX_PIXEL ) LP_accum = MAX_PIXEL;

    // Return approximation and detail coefficients
    return (HP_accum << 16) | LP_accum;
}
```

Compilation approach - Code example DES

```
    origin = (uint64_t)left_origin << 32 | right_origin;
    good = (uint64_t)left_good << 32 | right_good;

    if( MYTHREAD == 0 )
        printf("Nb keys by node : %ld\n", block_size);

    total = 0;

    upc_forall(block_key=0; block_key<nb_blocks; block_key++; block_key){
        upc_forall(core=0; core<6; core++; core)
            result[core] = des_breaker(block_key * block_size + core * (block_size / 6),
                    block_key * block_size + (core + 1) * (block_size / 6), origin, good);

        for(core=0; core<6; core++){
            if( result[core] != 0 ){
                printf("Key found : 0x%lx on thread %d, by the core %d\n", result, MYTHREAD, core);
                upc_global_exit(0);
            }
        }
    }

    printf("Thread n-%d, done\n", MYTHREAD);
    return 0;
}
```

Test bed

- Cray XD1
  - 12 AMD Opteron processors at 2.4 GHz
  - 6 Xilinx Vertex II FPGAs
  - Fast RapidArray interconnects
- Berkeley UPC
- Impulse C

Throughput and speedup over an Opteron 2.4GHz for the library approach on XD1

	Opteron 2.4GHz	1 FPGA	6 FPGAs
DWT (MB/s)	83	1 104 – (13.3×)	6 273 – (75.5×)
Sobel Edge detection (MB/s)	101	1 095 – (10.9×)	6 216 – (61.1×)
DES encryption (MB/s)	16	1 088 – (68.8×)	6 185 – (391.3×)
DES breaking (M keys/s)	2	1 194 – (600.×)	7 112 – (3 574.×)

Throughput and speedup over an Opteron 2.4GHz for the compilation approach on XD1

	1 FPGA, 1 engine/FPGA	1 FPGA, 6 engines/FPGA	6 FPGAs, 6 engines/FPGA
DWT	1/13.6	N/A	1/2.4
Sobel Edge detection	1/15.9	N/A	1/2.7
DES breaking	2.3	10.2	61.2

Conclusions

- We proposed a new programming model addressing entire HPRCs
- Two approaches:
  - a library approach *rightarrow* good performances
  - compilation approach *rightarrow* promising results
  - Opportunity to have a coherent workflow : C $\longrightarrow$ UPC $\longrightarrow$ UPC version targeting FPGAs$\longrightarrow$ Optimized core in HDL

Future Work

- New platform will be investigated
- Evaluate other parallel languages: Sequoia, Chaptel, X10, ...
- Furthermore adapting the programming model (memory model, ...)
- Extention to other types of accelerators (Cell, GPU, ...)

Questions