Masterpraktikum Scientific Computing

High-Performance Computing

Michael Bader
Alexander Heinecke

Technische Universität München, Germany
Outline

Intel Cilk Plus

OpenCL
Intel Cilk Plus in a nutshell

- new parallel programming model based on three keywords
- support for reductions and hyper-objects (not covered!)
- same functionality like OpenMP task
- including vector extensions as alternative to #pragma simd
Cilk Keywords

The three Cilk Keywords are

- _Cilk_spawn
- _Cilk_sync
- _Cilk_for

In order to use the simpler versions (cilk_spawn, clik_sync, cilk_for), you have to include:
<cilk/cilk.h>

By exporting the CILK_NWORKERS variable the number of workers can be set!
Thread Spawning and Synchronization

Definition:

```c
var = cilk_spawn func(args);
cilk_spawn func(args);
```

Definition:

```c
cilk_sync; // only syncs with children spawned
    // by this function. Children
    // of other functions are not affected.
```
Loop-Parallelization

Usage:

cilk_for (int i = begin; i < end; i+=2)
  f(i);
cilk_for (T::iterator i(vec.begin()); i!=vec.end(); ++i)
  g(i);

Be aware of the grain size and the parallelization method used by the Cilk runtime!
Example

```c
void traverse(node* current)
{
    cilk_spawn traverse(current->getLeftChild);
    traverse(current->getRightChild);
    cilk_sync;
}
```
Array Notations

Extensions for Array Notations Programming Model offer an additional high-level access to the Intel compiler’s vectorizer!

section operator ::= [<lower bound> : <length> : <stride>]

Examples: (taken from the compiler’s user guide)

a[0:3][0:4] // refers to 12 elements in the
  // two-dimensional array a, starting
  // at row 0, column 0, and
  // ending at row 2, column 3.

b[0:2:3] // refers to elements 0 and 3 of the
  // one-dimensional array b

b[::] // refers to the entire array b
Operator and Assignment Maps

Operator Maps: (taken from the compiler’s user guide)

\[ a[::] * b[::] \quad // \text{element-wise multiplication} \]

\[ a[3:2][3:2] + b[5:2][5:2] \quad // \text{matrix addition of} \]
\[ \quad // \text{2x2 matrices in a and} \]
\[ \quad // \text{b starting at a[3][3]} \]
\[ \quad // \text{and b[5][5]} \]

Assignment Maps: (taken from the compiler’s user guide)

\[ a[::][::] = b[::][2][::] + c; \]
\[ e[::] = d; \]
Reductions

built-in reductions:

```python
__sec_reduce_add(a[:])
__sec_reduce_mul(a[:])
__sec_reduce_all_zero(a[:])
__sec_reduce_all_nonzero(a[:])
__sec_reduce_any_nonzero(a[:])
__sec_reduce_min(a[:])
__sec_reduce_max(a[:])
__sec_reduce_min_ind(a[:])
__sec_reduce_max_ind(a[:])
```

**Note:** there is a possibility for user-defined reductions!
A note on dynamically sized multi-dimensional arrays

Normally, a multi-dimensional array is allocate as a one-dimensional array:
```c
double* a = (double*)malloc(x*y*sizeof(double));
```

This leads to element accesses like:
```c
double foo = a[i*y+j];
```

In case of Cilk Plus we need n-d access, so we specify
```c
double** a_2d = (double**)malloc(x*sizeof(double*));
a_2d[0:x] = &(a[0:x:y]);
```

and can access foo by
```c
double foo = a_2d[i][j];
```
OpenCL in a nutshell

- open standard that supports a wide range of data-parallel compute devices
- Backends by: NVIDIA (GPU), Intel (CPU), AMD (GPU, CPU), IBM (Power, Cell BE)
- Code is generated at runtime
- somehow inspired by CUDA
- current release is 1.2, implemented 1.1
- OpenCL offers portability but not performance portability (same problem as with MPI! :-() 
- several OpenCL backends can be installed in parallel (selection happens at runtime)

including OpenCL + adjusting compiler’s include path (if needed):
#include "CL/cl.h"
Steps required to execute an OpenCL kernel

- select platform (usually vendor)
- select device (CPU / GPU) and create context
- create command queue for device
- build program/kernels for device
- create buffers for data being transferred to kernels
- enqueue kernels and start kernels
- transfer result to host
- shutdown

This code is often called *boiler-plate code*!

Introduction:

Parallelization

source: AMD
Memory Hierarchy

source: AMD
Selecting Platform and Device

// determine number of available OpenCL platforms
err = clGetPlatformIDs(0, NULL, &num_platforms);
if (err != CL_SUCCESS) { /* HANDLE! */ }

// get available platforms
platform_ids = new cl_platform_id[num_platforms];
err = clGetPlatformIDs(num_platforms, platform_ids, NULL);
if (err != CL_SUCCESS) { /* HANDLE! */ }

for (cl_uint ui = 0; ui < num_platforms; ui++)
{
    char vendor_name[128] = {0};
    err = clGetPlatformInfo(platform_ids[ui], CL_PLATFORM_VENDOR,
                128 * sizeof(char), vendor_name, NULL);
    if (err != CL_SUCCESS) { /* HANDLE! */ }
    if (strcmp(vendor_name, "Intel(R) Corporation") == 0)
        platform_id = platform_ids[ui];
}

device_ids = new cl_device_id[1]; // only 1 device
cl_uint num_devices = 1;
err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_CPU, 1,
    device_ids, &num_devices);
if (err != CL_SUCCESS) { /* HANDLE! */ }

context = clCreateContext(0, 1, device_ids, NULL, NULL, &err);
Creating Command-Queue and Building Kernel for Device

```c
command_queue = clCreateCommandQueue(context, device_ids,
    CL_QUEUE_PROFILING_ENABLE, &err);
if (err != CL_SUCCESS) { /* HANDLE! */ }

const char* kernel_src = ...;
program = clCreateProgramWithSource(context, 1, &kernel_src, NULL, &err);
if (err != CL_SUCCESS) { /* HANDLE! */ }

err = clBuildProgram(program, 0, NULL, "-cl-finite-math-only
    -cl-strict-aliasing -cl-fast-relaxed-math
    -cl-single-precision-constant", NULL, NULL);
if (err != CL_SUCCESS) { /* HANDLE! */ }

kernel = clCreateKernel(program, "kernel", &err);
if (err != CL_SUCCESS) { /* HANDLE! */ }
```
Creating Buffers and Starting Kernels through Command-Queues and Shutdown

```c
in = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
            sizeof(float)*in_length, data, NULL);
out = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
            sizeof(float)*out_length, NULL, NULL);

if ( clSetKernelArg(kernel, 0, sizeof(cl_mem), &in) ||
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &out) ) != CL_SUCCESS)
    { /* HANDLE! */ }

size_t global = ...; size_t local = ...; size_t dim = ...;
err = clEnqueueNDRangeKernel(command_queue, kernel, dim, NULL, &global,
        &local, 0, NULL, NULL);
if (err != CL_SUCCESS) /* HANDLE! */

err = clEnqueueReadBuffer(command_queue, out, CL_FALSE, 0,
            sizeof(float)*out_length, data_back, 0, NULL, 0);
if (err != CL_SUCCESS) /* HANDLE! */

clReleaseMemObject(in);
clReleaseMemObject(out);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(command_queue);
clReleaseContext(context);
```
Kernel Examples

```c
void add (const float* src_a,
         const float* src_b,
         float* res,
         const int num)
{
    for (int i = 0; i < num; i++)
        res[i] = src_a[i] + src_b[i];
}

__kernel void add_opencl (__global const float* src_a,
                         __global const float* src_b,
                         __global float* res)
{
    const int idx = get_global_id(0);
    res[idx] = src_a[idx] + src_b[idx];
}
```
Hints

- several devices can be used by duplicating kernels, command-queues and buffers; usually just arrays of them and iterating over them, see example in NVIDIA SDK!
- work-group size should be a multiple of 32 or 64 in order to ensure best performance on GPUs and CPUs!
- small loops inside kernels should be avoided (poor performance on GPUs)
- due to runtime-compilation, such loops can be fully unrolled, as you can generate OpenCL code on the fly (e.g. stringstreams in C++)!
- use the max. possible dimension of your work-group grid in order to increase parallelism!
- avoid branches when- and wherever you can!