Can we overcome the performance/portability tradeoff on GPU pipelines?

Cyril CETRE
PhD student at Thales Research and Technology & Observatoire de Paris
Accelerators and time sensitive Cyber-Physical Systems (CPS)
General purpose GPUs are

- Ever-growing high-throughput accelerators
- Providing an increasingly mature ecosystem with affiliated APIs & dedicated hardware

They’re not:

- Providing, out-of-the-box, means to prioritize deadlines over throughput (yet)
- Effortless to integrate to time-sensitive cyber-physical systems
Use case CPS – Micado HRTC

Hardware diversity implies:

- Data movement
- Hardware Synchronisation
- Monitoring

Focus of this presentation
Complex pipelines & GPU workloads introduce jitter inherent to GPU computing

It is possible to reduce jitter using GPU synchronization strategy

This is a solution proposed by the Observatoire de Paris for AO RTC
**GPU - time-sensitive computing**

Schematic representation of an application critical path over time using Naive and GPU sync strategy

- **Naive**
- **GPU sync strategy**

- **Stream 1**
- **Stream 2**

- **Sensor**
- **Signal**
- **Computation**
- **GPU sync Notification**
- **Busy-wait**
Results obtained with a simplified RTC

Time to solution histogram and execution profile using 2 different synchronization mechanisms over 1M iterations
GPUs – design challenges

- Out of the original software design for the GPU, tricky to maintain
- New GPUs vendors coming, does it imply new development process?
- Can we decrease response time variability yet again?
OpenMP API: Parallel programming for productivity
OpenMP - introduction

Mature language constantly reviewed (last release Nov 2021, v5.2)

> One of industrial standards in HPC for shared-memory systems
> Active research community with an increasing interest on the embedded domain
> Barcelona Supercomputing Center has a leading research position on that matter

Productivity

> performance
  - Support for different types parallelism and accelerator devices.
> Portability
  - Supported by many chip vendors (Intel, IBM, ARM, NVIDIA, TI, Gaisler, Kalray).
> Programmability
  - Interoperability with other programming models (e.g., CUDA, OpenCL).
  - Allows incremental parallelization and can be easily compiled sequentially.
Barcelona Supercomputing Center - OpenMP tasking model

Sequential version
void main() {
    int x,y;
    f1(&x,&y);
    f2(x);
    f3(y);
}

OpenMP version
void main() {
    #pragma omp parallel
    #pragma omp single
    {
        int x,y;
        #pragma omp task depend(out:x,y)
        {
            f1(&x,&y);
        }
        #pragma omp task depend(in:x)
        {
            f2(x);
        }
        #pragma omp target map(to:y) depend(in:y)
        {
            f3(y);
        }
    }
}

1. Open parallelism

2. Tasks executed on the host

3. Tasks executed on the host and accelerator when f1 completes

Slide credit : Eduardo Quiñones – Barcelona Supercomputing Center
Barcelona Supercomputing Center – OpenMP GPU Graph model

Chenile, Y., Royuela, S., and Quiñones, E. OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices, in SCOPES. 2020.
Introducing GPU graphs
GPU graphs - Introduction

- Available using Nvidia and AMD ecosystems
- Define a directed acyclic graph of kernels
- Greatly reduce kernels overheads.
GPU graphs – a powerful tool to reduce time variability

Without graphs

With graphs
GPU graphs – Theoretical performance

Time to solution histogram and execution profile using 3 different synchronization mechanisms over 1M iterations on a benchmark application

<table>
<thead>
<tr>
<th></th>
<th>MET (μs)</th>
<th>WCET (μs)</th>
<th>RMS jitter (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU sync</td>
<td>384.1</td>
<td>400.3</td>
<td>5.0</td>
</tr>
<tr>
<td>GPU sync &amp; graph</td>
<td>326.2</td>
<td>331.8</td>
<td>1.1</td>
</tr>
<tr>
<td>CPU sync</td>
<td>466.2</td>
<td>482.6</td>
<td>1.6</td>
</tr>
</tbody>
</table>
Event-based OpenMP synchronizations
Proposal: event-based OpenMP task

Runtime-based control loop*

OpenMP runtime support

```c
#pragma omp parallel
#pragma omp single nowait
{
#if pragma omp task event(periodic:100)  
rt_task_1();
#else pragma omp task event(sporadic:event1)  
rt_task_N();
#endif pragma omp task event(sporadic:event2)  
rt_task_N();
}
```
OpenMP – Main contributions

User code

```c
#pragma omp taskgraph tdg_type(static)
{
    // undef /*BSC_OPENMP*/

    int calibOut, intensityOut, reduceOut, reduceOut2, maskedPix;

#pragma omp target nowait depend (inout : calibOut) ksize(32, 256, 0) activewait(d_syncData)
{
    __kernel_calib(d_img_raw, d_img, d_dark, d_flat, d_lutPix, IMG_SIZE);
}

#pragma omp target nowait depend (in : calibOut) depend (out : intensityOut) ksize(32, 256, 0)
{
    __kernel_fillIntensities(d_intensities, d_img, d_subindex, d_subindex, NTOT, NSLOPES);
}

#pragma omp target nowait depend (in : intensityOut) depend (out : reduceOut) ksize(96, 256, 1024)
{
    __kernel_reduce3(d_intensities, d_block_sums, NSLOPES);
}

#pragma omp target nowait depend (in : reduceOut) depend (out : reduceOut2) ksize(1, 256, 1024)
{
    __kernel_reduce2(d_block_sums, d_sumIntensities, GRIDSZ);
}
```

generated

```c
cudaGraphNode_t kernelNode_0;
cudaKernelNodeParams kernelParams_0;
kernelparams_0.func = (void *)__kernel_calib;
kernelparams_0.gridDim = dim3(32,1,1);
kernelparams_0.blockDim = dim3(256,1,1);
kernelparams_0.sharedMemBytes = 0;
kernelparams_0.extra = NULL;

void *__kernelArgs_0[6];
char *ptr_0_0 = (char *)__d_ptr_d_img_raw + 0 * 4;
kernelparams_1[0] = (void *)&ptr_0_0;
char *ptr_0_1 = (char *)__d_ptr_d_img + 0 * 4;
kernelparams_1[1] = (void *)&ptr_0_1;
char *ptr_0_2 = (char *)__d_ptr_d_dark + 0 * 4;
kernelparams_1[2] = (void *)&ptr_0_2;
char *ptr_0_3 = (char *)__d_ptr_d_flat + 0 * 4;
kernelparams_1[3] = (void *)&ptr_0_3;
char *ptr_0_4 = (char *)__d_ptr_d_lutPix + 0 * 4;
kernelparams_1[4] = (void *)&ptr_0_4;
int constant_0_5 = 57600;
kernelparams_1[5] = (void *)&constant_0_5;
kernelparams_0.kernelParams = (void **)kernelArgs_1;
deps.clear();
//ddeps.push_back (kernelNode_0 llvm);
cudagraphAddKernelNode (&kernelNode_0, graph, deps.data(),
deps.size(), &kernelParams_0);
```
Results with an AO RTC pipeline

Simplified code:

```
#pragma omp parallel
#pragma omp single
for (n_iters)
    #pragma omp taskgraph
    {
        #pragma omp task depend(...) WFS_camera();
        #pragma omp task depend(...) pixel_processing();
        #pragma omp task depend(...) RTR_MVM();
    }
```
Takeaways

- A new extension to OpenMP tasking model
- Available for both AMD ROCm and Nvidia CUDA accelerators
- In our use cases, we never exceeded 10μs of RMS jitter and 30μs of max jitter
- A lot of room for improvement (conditional graphs, device launched graphs)