MODELING CUDA COMPUTE APPLICATIONS BY CRITICAL PATH

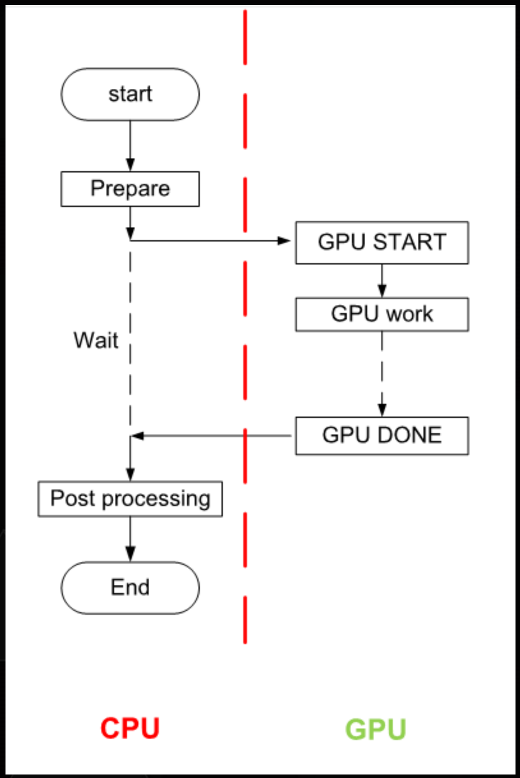
PATRIC ZHAO, JIRI KRAUS, SKY WU
patricz@nvidia.com

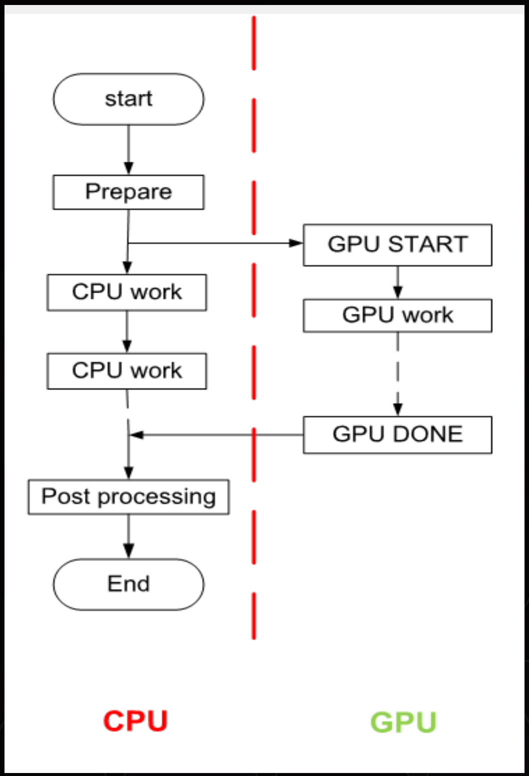
AGENDA

- Background
- Collect data and Visualizations
- Critical Path
- Performance analysis and prediction
  - Case Study from GROMACS

1.BACKGROUND

- Heterogeneous Computing





GPU TECHNOLOGY CONFERENCE

Toy Example:

```
int main()
{
   const int num_threads = 2;

   pthread_t threads[num_threads];

   // Launch GPU Async Work
   if (pthread_create(&threads[0], NULL, launch_GPU_work, 0)) {
      fprintf(stderr, "Error creating threadn");
      return 1;
   }

   // Launch CPU Async Work
   if (pthread_create(&threads[1], NULL, launch_CPU_work, 0)) {
      fprintf(stderr, "Error creating threadn");
      return 1;
   }

   // Wait Results
   for (int i = 0; i < num_threads; i++) {
      if(pthread_join(threads[i], NULL)) {
         fprintf(stderr, "Error joining thread %n“, i);
         return 2;
      }
   }

   return 0;
}
```

```
void *launch_GPU_work(void *dummy)
{
 float *data_d = NULL, *data_h = NULL;
 int memsize = 24*N*sizeof(float);

 data_h = (float*)malloc(memsize);
 memset(data_h, 0, memsize);
 cudaMalloc(&data_d, memsize);

 cudaMemcpy(data_d, data_h, memsize, cudaMemcpyHostToDevice );

 gpu_work<<<1, 64>>>(data_d, 24*N);
 cudaStreamSynchronize(0);

 cudaMemcpy(data_h, data_d, memsize, cudaMemcpyDeviceToHost);
 return NULL;
}

__global__ void gpu_work(float *x, int n)
{
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   for (int i = tid; i < n; i += blockDim.x * gridDim.x) {
      x[i] = sqrt(pow(3.14159,i));
   }
}
```

GPU TECHNOLOGY CONFERENCE

```c
int main()
{
    const int num_threads = 2;

    pthread_t threads[num_threads];

    // Launch GPU Async Work
    if (pthread_create(&threads[0], NULL, launch_GPU_work, 0)) {
        fprintf(stderr, "Error creating threadn");
        return 1;
    }

    // Launch CPU Async Work
   if (pthread_create(&threads[1], NULL, launch_CPU_work, 0)) {
       fprintf(stderr, "Error creating threadn");
       return 1;
   }

   // Wait Results
   for (int i = 0; i < num_threads; i++) {
       if(pthread_join(threads[i], NULL)) {
           fprintf(stderr, "Error joining thread %n“, i);
           return 2;
       }
   }

    return 0;
}
```

```c
void *launch_CPU_work(void *dummy)
{
    float *data;
    data = (float*) malloc(4*N*sizeof(float));
    cpu_work(data, N*4);
    return NULL;

}

void cpu_work(float *x, int n)
{
   for(int i = 0; i < n; i++) {
       x[i] = sqrt(pow(3.14159,i));
   }
}
```

In this example:

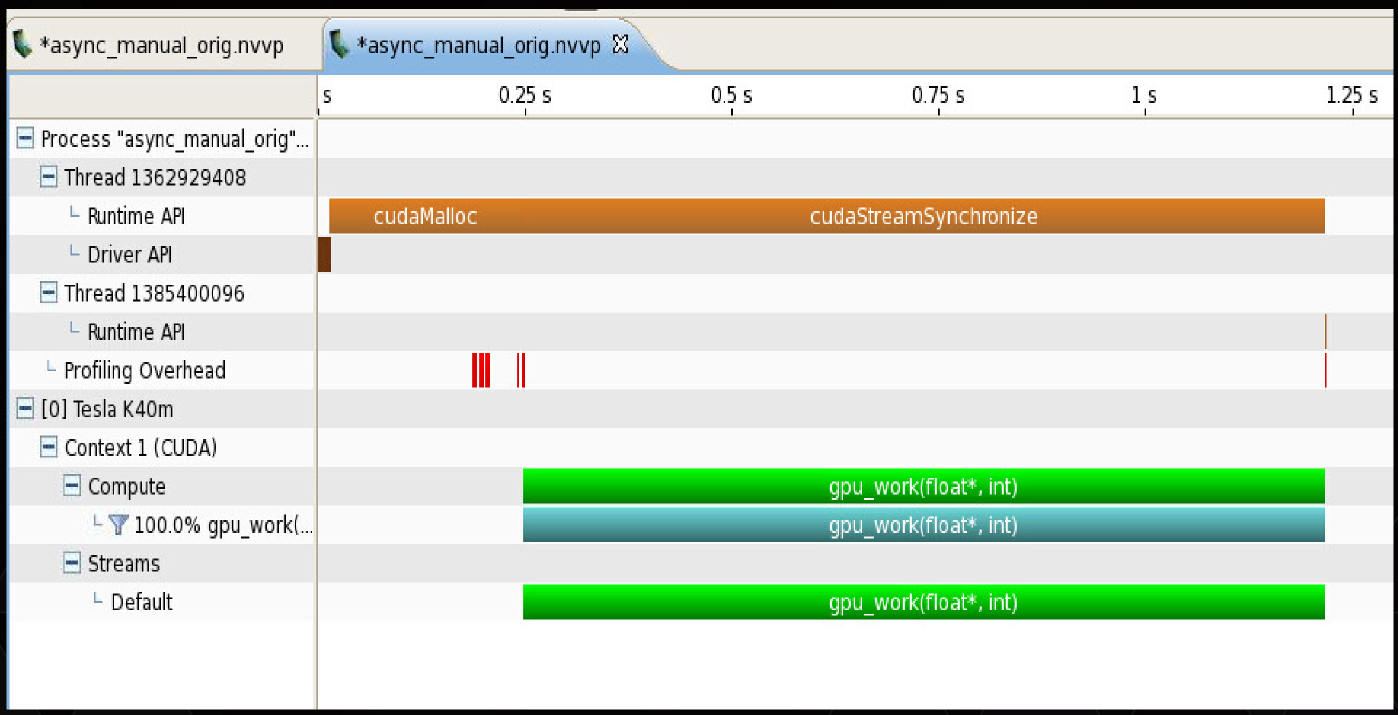
a) GPU may run longer and dominate the runtime

b) CPU may run longer and dominate the runtime

c) CPU and GPU overlap each other

So, we need to triage from a system perspective both CPU and GPU

But, we run with NV Visual Profiler directly, we only can get GPU runtime.



2.COLLECT DATA AND VISUALIZATION

➢Generate Custom Application Profile Timelines

NVTX , NVIDIA Tools Extension :

Application level APIs for NVIDIA Profiler tools

➢ How to use it ?

1) Add time stamps manually
2) Use compiler instrumentation automatically

Details in Jiri's blog: customer profiler timelines

```
#ifdef USE_NVTX
#include "nvToolsExt.h"
const uint32_t colors[] = { 0x0000ff00, 0x000000ff, 0x00ffff00,
                            0x00ff00ff, 0x0000ffff, 0x00ff0000, 0x00ffffff };
const int num_colors = sizeof(colors)/sizeof(uint32_t);

#define PUSH_RANGE(name,cid) { \
int color_id = cid; \
color_id = color_id%num_colors;\
nvtxEventAttributes_t eventAttrib = {0}; \
eventAttrib.version = NVTX_VERSION; \
eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE; \
eventAttrib.colorType = NVTX_COLOR_ARGB; \
eventAttrib.color = colors[color_id]; \
eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII; \
 eventAttrib.message.ascii = name; \
nvtxRangePushEx(&eventAttrib); \ }

#define POP_RANGE nvtxRangePop();
#else
#define PUSH_RANGE(name,cid)
#define POP_RANGE
#endif
```

GPU TECHNOLOGY CONFERENCE

Add time stamps manually

a) Define MACROS
b) Add MACROS from source code
c) Compile with -lnvToolsExt
d) Run binary and generate data from NVCC

```
void *launch_CPU_work(void *dummy)
{
   PUSH_RANGE("prepare_CPU_work",1)
   float *data;
   data = (float*) malloc(36*N*sizeof(float));
   POP_RANGE

   PUSH_RANGE("cpu_work", 2)
   cpu_work(data, N*36);
   POP_RANGE
   return NULL;
}
```

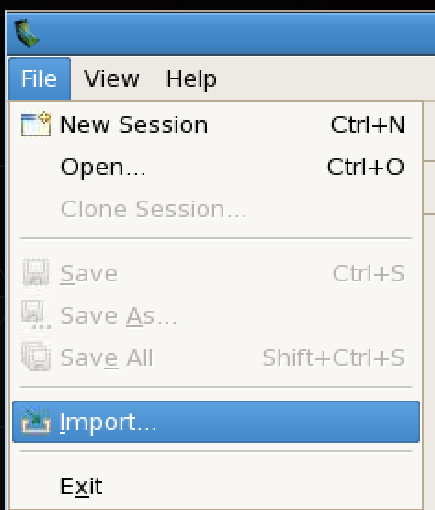
Example CMD:

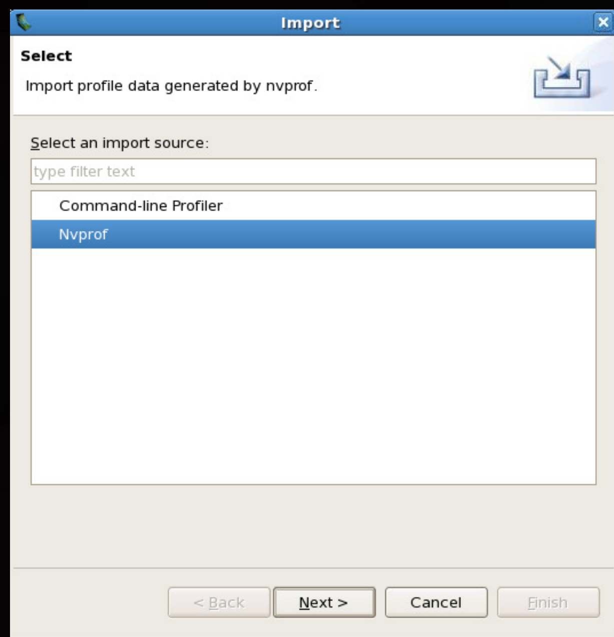
```
nvcc -DUSE_NVTX
     -arch=sm_35
     -lnvToolsExt
     -o manual_nvtx manual_nvtx.cu
```

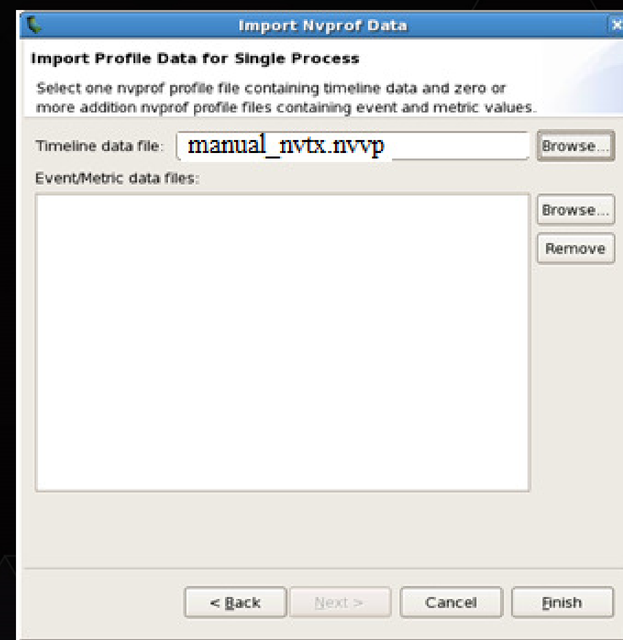
GPU TECHNOLOGY CONFERENCE

VISUALIZE RESULTS

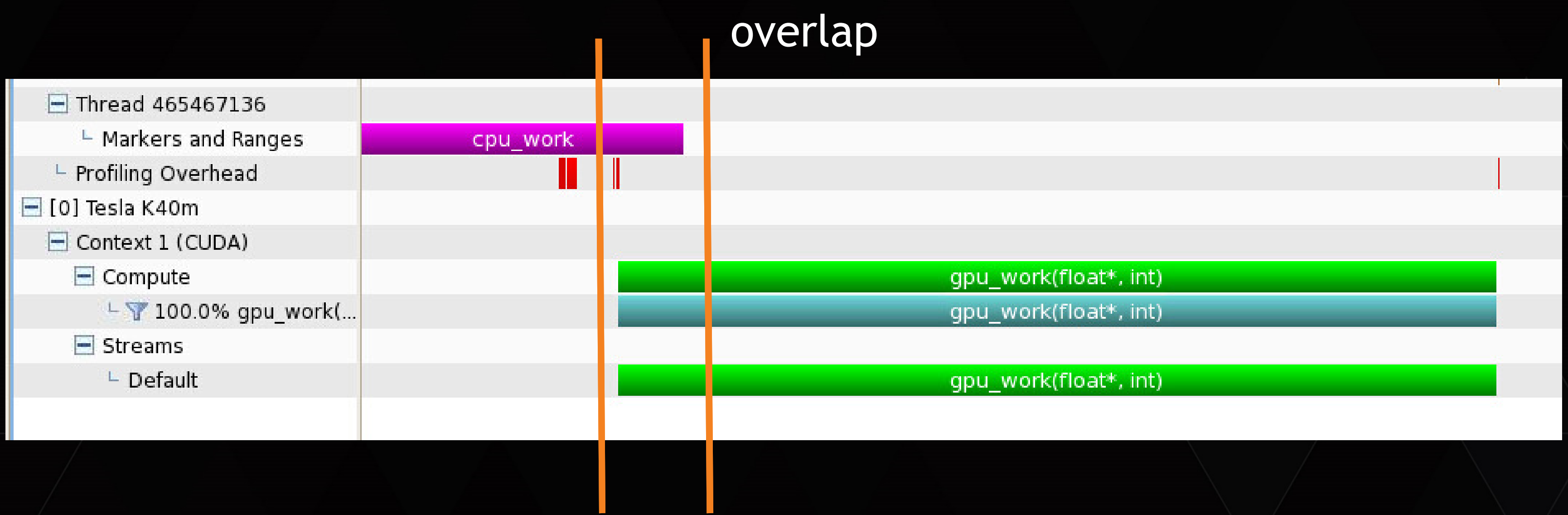
- NVidia Visual Profiler (NVVP)
  - *nvprof -o manual_nvtx.nvvp ./manual_nvtx*



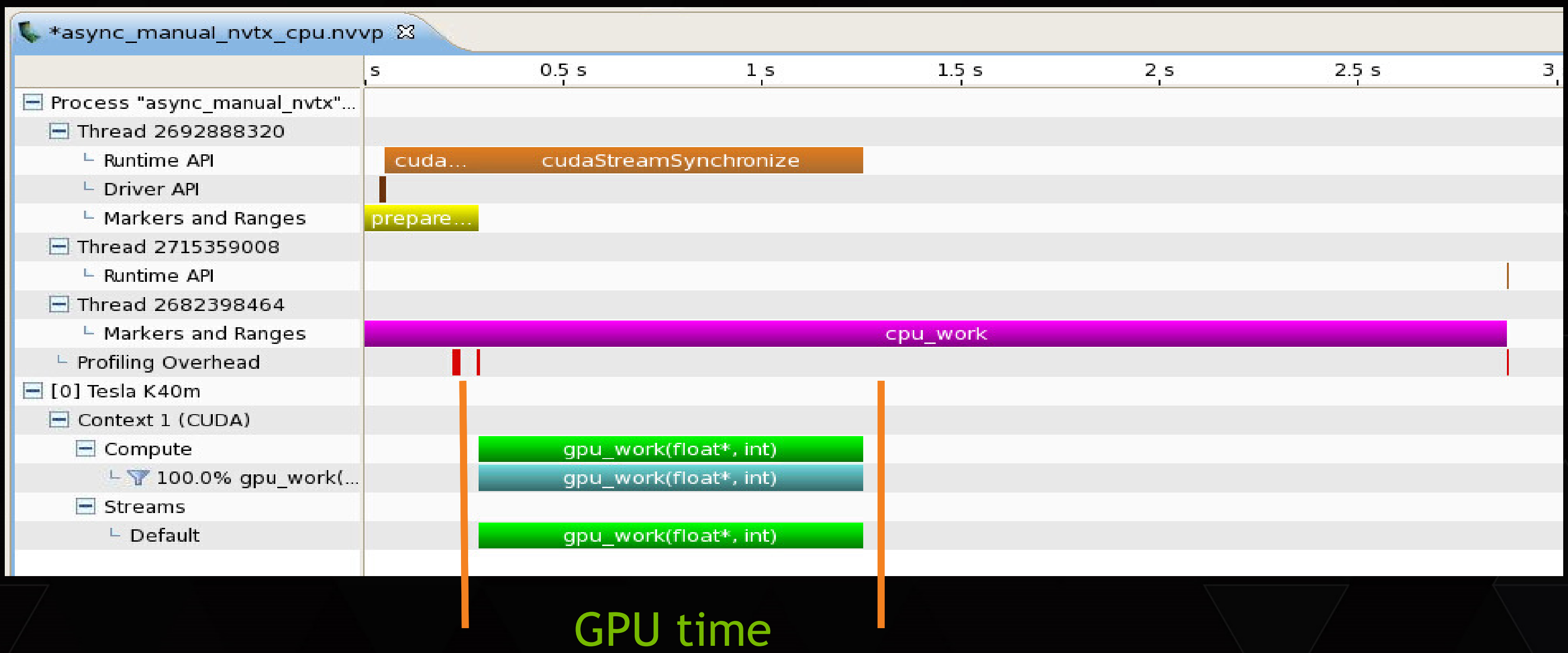




- Results from our toy example:



In this situation, GPU execution time is the main part of total runtime, our performance optimization can start from GPU kernel!



In this situation, GPU execution time is hidden under CPU timeline, so we don’t need care about GPU performance anymore. Instead, we need to optimize CPU task.

GPU TECHNOLOGY CONFERENCE

GET RAW DATA FOR CUSTOMER ANALYSIS

nvprof --print-api-trace --print-gpu-trace --csv
--log-file counter.txt ./manual_nvtx

timeline and duration

```
==1929== NVPROF is profiling process 1929, command: ./async_manual_nvtx
==1929== Profiling application: ./async_manual_nvtx
==1929== Profiling result:
```

Start	Duration	Grid Size	Block Size	Regs*	SSMem*	DSMem*	Size	Throughput	Device	Context	Stream	Name
199.85ms	-	-	-	-	-	-	-	-	-	-	-	[Range start] prepare_CPU_work
199.86ms	-	-	-	-	-	-	-	-	-	-	-	[Range end] prepare_CPU_work
199.86ms	-	-	-	-	-	-	-	-	-	-	-	[Range start] cpu_work
199.89ms	-	-	-	-	-	-	-	-	-	-	-	[Range start] prepare_GPU_work
....	....											
524.32ms	288.31ms	-	-	-	-	-	-	-	-	-	-	cudaMalloc
812.63ms	11.001ms	-	-	-	-	-	-	-	-	-	-	cudaMemcpy
812.76ms	10.959ms	-	-	-	-	-	100.66MB	9.1855GB/s	Tesla K40m (0)	1	7	[CUDA memcpy HtoD]
823.64ms	-	-	-	-	-	-	-	-	-	-	-	[Range end] prepare_GPU_work
823.64ms	-	-	-	-	-	-	-	-	-	-	-	[Range start] GPU_work
823.65ms	1.4960us	-	-	-	-	-	-	-	-	-	-	cudaConfigureCall
823.65ms	8.1360us	-	-	-	-	-	-	-	-	-	-	cudaSetupArgument
823.66ms	203ns	-	-	-	-	-	-	-	-	-	-	cudaSetupArgument
823.66ms	39.545us	-	-	-	-	-	-	-	-	-	-	cudaLaunch (gpu_work(float*, int) [356])
823.70ms	984.88ms	-	-	-	-	-	-	-	-	-	-	cudaStreamSynchronize
823.73ms	984.85ms	(1 1 1)	(64 1 1)	20	0B	0B	-	-	Tesla K40m (0)	1	7	gpu_work(float*, int) [356]
1.80858s	-	-	-	-	-	-	-	-	-	-	-	[Range end] GPU_work
1.80858s	11.631ms	-	-	-	-	-	-	-	-	-	-	cudaMemcpy
1.80860s	11.434ms	-	-	-	-	-	100.66MB	8.8039GB/s	Tesla K40m (0)	1	7	[CUDA memcpy DtoH]
2.49997s	-	-	-	-	-	-	-	-	-	-	-	[Range end] cpu_work

CPU work start

GPU info

CPU work end

GPU TECHNOLOGY CONFERENCE

Customer Plotting for Publication-quality graphs

- Clean data : start time, end time, duration

CPU: get information from NVTX;

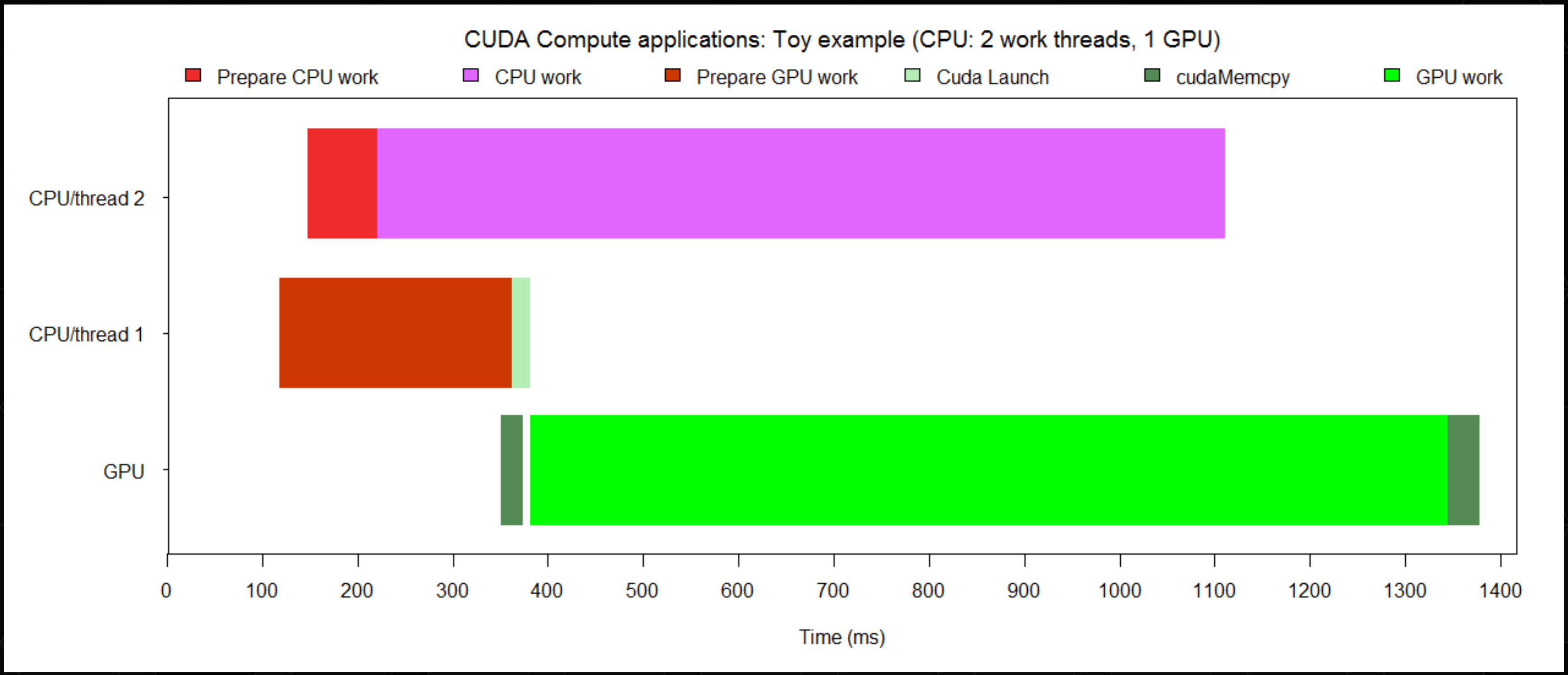
GPU: nvprof can dump GPU API automatically,
such memory copy, kernel launch

Name	Start	End	Duration	Device
Prepare GPU work	118	362	244	thd1
Prepare CPU work	148	220	72	thd2
CPU work	221	1100	879	thd2
Cuda Memcpy HtoD	350	362	12	GPU
Cuda Launch	362	370	8	thd1
GPU work	371	1344	963	GPU
Cuda Memcpy DtoH	1345	1367	22	GPU

➢ Plotting by Gantt-Chart

In this session, I plot by R, and you can use any tools you like

Excel, Python, Matlab, gnuplot …

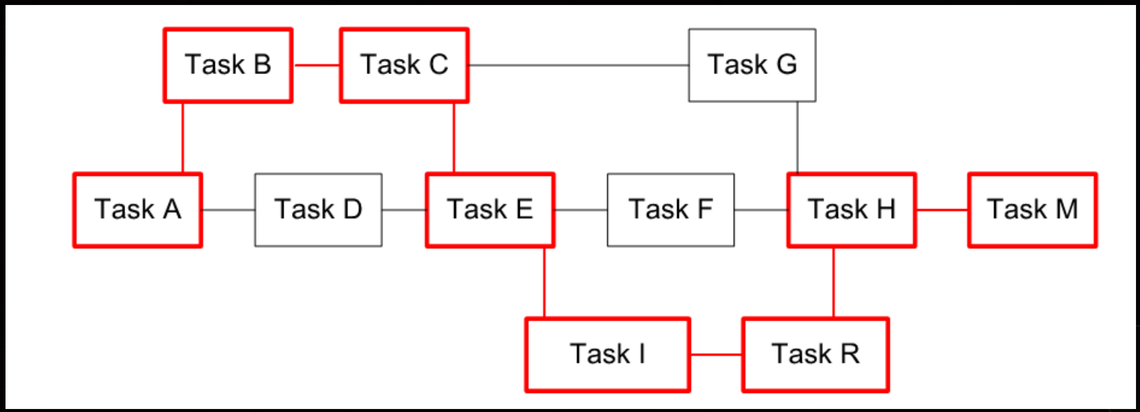


GPU TECHNOLOGY CONFERENCE

3. CRITICAL PATH

➢What's critical path ?

Simple : the longest duration path of tasks that leads through the project.



➢ Why we care about it ?

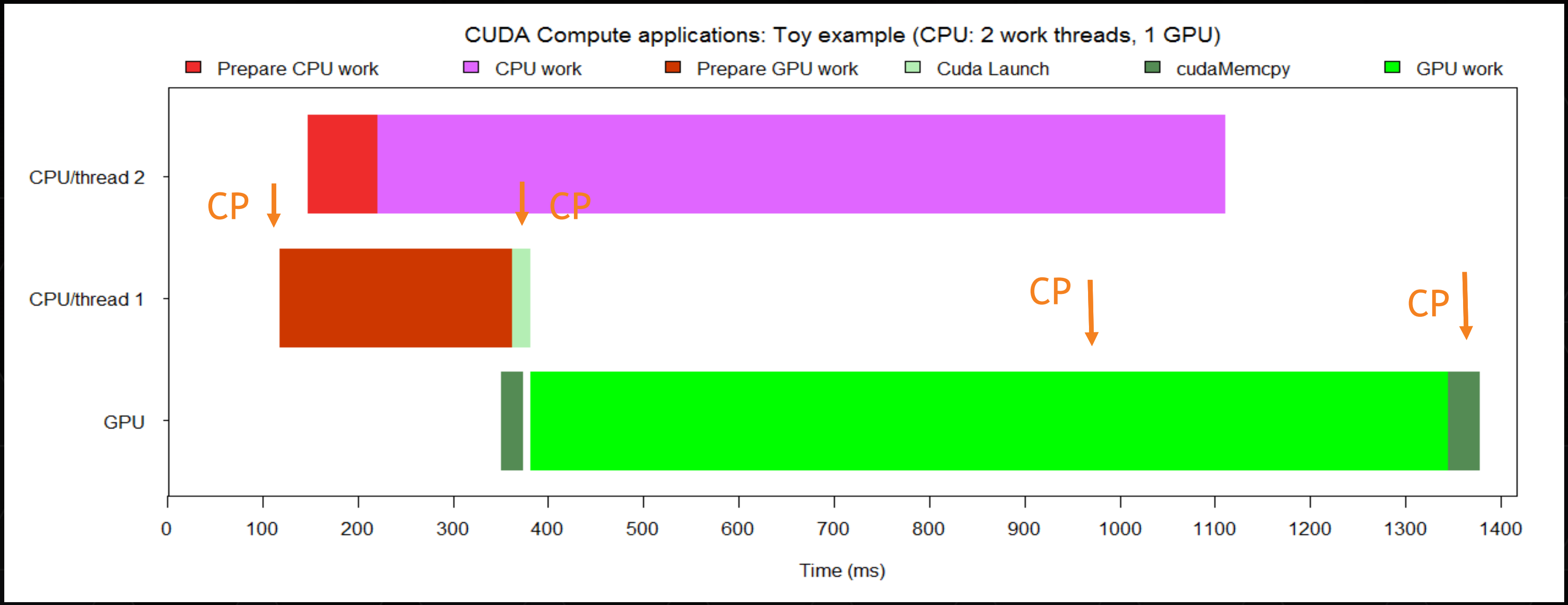
If delay occurs on critical path, the whole project will be delay;

So, our optimization should start from critical path.

GPU TECHNOLOGY CONFERENCE

➢How to identify critical path?

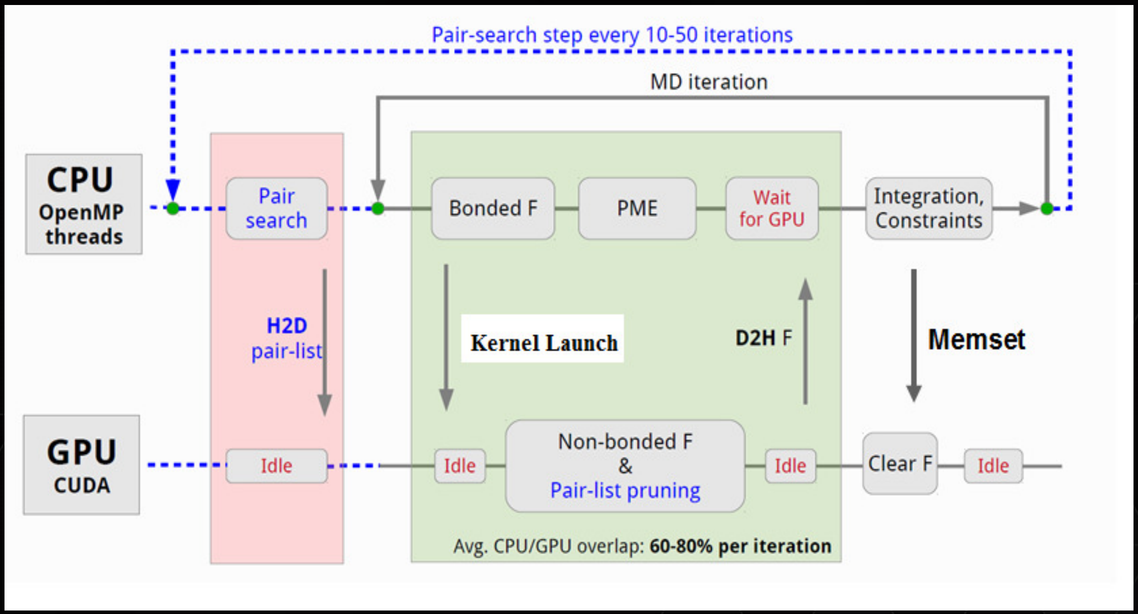
1) start from end point of timeline
2) label CP (critical path) for current function
3) backward to it's previous function (caller w/ priori)
4) go to step 2 until start point



4.PERFORMANCE ANALYSIS AND PREDICTION

Case Study from GROMCS which is a molecular dynamics program

Heterogeneous computation diagram



Szilárd Páll, GTC2013, Challenges and Solution for Heterogeneous Parallelization of Molecular Dynamics at 10,000 fps

CASE 1: Performance Analysis

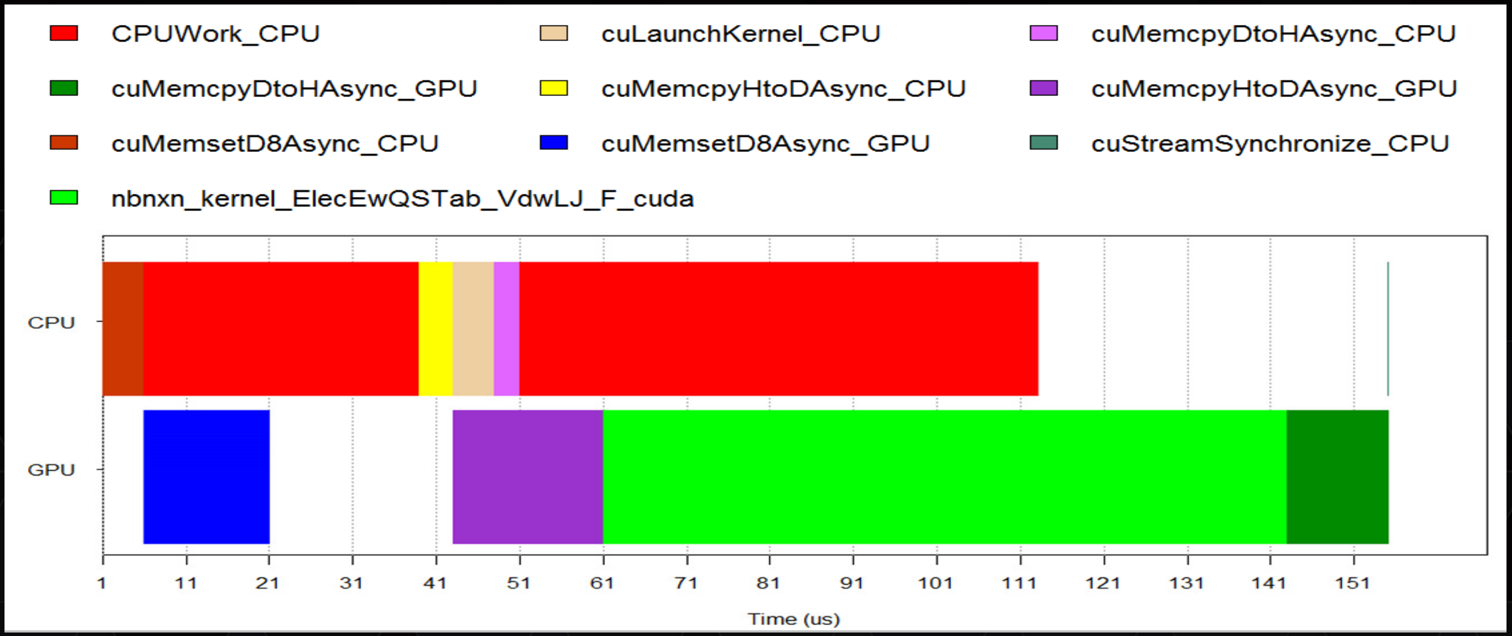
Target:

How much CUDA API launch time delay the whole application for small benchmark ?

ANALYSIS WITHOUT CRITICAL PATH

- - Measured the whole application time
- - Measured CUDA API launch time

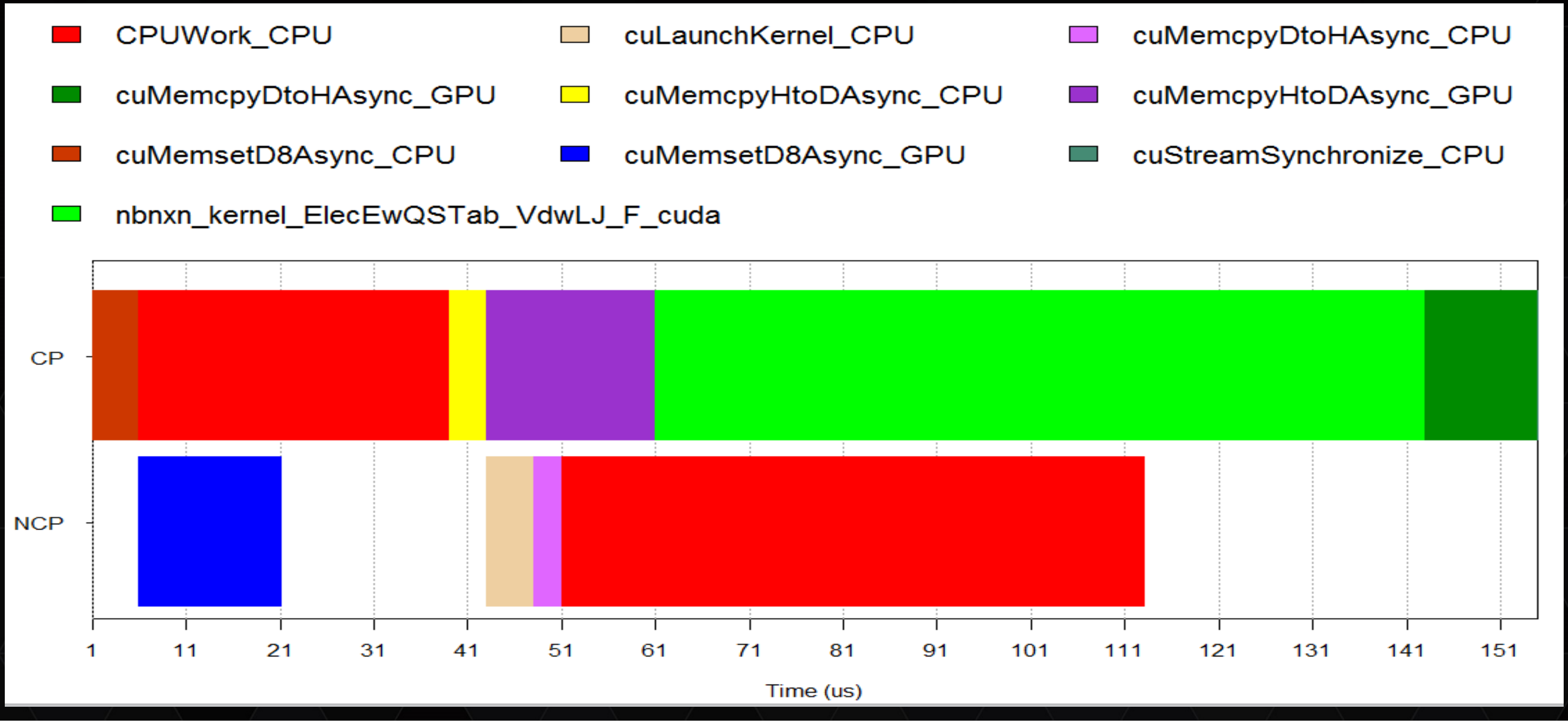
  4 times cuda API async launches from CPU
- - About 15% ($\sum CudaAPILaunchTime$ / TotalTime)



ANALYSIS WITH CRITICAL PATH

Identify Critical Path

- 2 Async Calls on Critical Path
- 2 Async Calls NOT on Critical Path



So, launch latency is not as crucial as previous analysis, and we can degrade it's priority for deep optimizations.

Critical Path Info	NO	Yes	
cudaMemsetAsync	Yes	Yes (CP)	4%
cudaMemcpyHtoDAsync	Yes	Yes (CP)	3%
cudaMemcpyDtoHAsync	Yes	No (NCP)	4%
cudaKernelLaunch	Yes	No (NCP)	3%
Total	15%	7%	

CASE 2: Performance Prediction

GPU TECHNOLOGY CONFERENCE

RNASE benchmark: Tested on GTX680 (GK104)

GPU kernel is the longest part of whole program.

Tested on GTX 680			
API name	start	duration	where
cudaStreamSynchronize_CPU	0	0	CPU
cudaMemsetD8Async_CPU	0	7	CPU
cudaMemsetD8Async_GPU	7	20	GPU
CPUWork_CPU	7	266	CPU
cudaMemcpyHtoDAsync_CPU	273	4	CPU
cudaMemcpyHtoDAsync_GPU	277	60	GPU
cudaLaunchKernel_CPU	277	7	CPU
cudaMemcpyDtoHAsync_CPU	284	4	CPU
CPUWork_CPU	288	900	CPU
nbnxn_kernel_ElecEwQSTab_VdwLJ_F_cuda	337	1165	GPU
cudaMemcpyDtoHAsync_GPU	1502	66	GPU
cudaStreamSynchronize_CPU	1568	0	CPU

Predict on K40X (or new algorithm)

- Tested Memory Bandwidth by SDK/bandwidthTest

GPU	GTX 680 (GK104)	K40X (GK110)
SMX	8	15
GPU clk	1006 MHz	875 MHz (boost)
Pinned HtoD	4.5 GB/sec	9.98 GB/sec
Pinned DtoH	3.9 GB/sec	10 GB/sec

GPU TECHNOLOGY CONFERENCE

Prediction

➢ Computation

- Estimate GPU time (assume scale-up linear):

  GTX / K40X = 8SM * 1006 MHz / 15SM * 875 MHz = 0.6

  → 1165 * 0.6 = 699 on K40X

➢ Memory

HtoD : 4.5 / 9.98 * 60 → 27

DtoH : 3.9 / 10 * 66 → 26

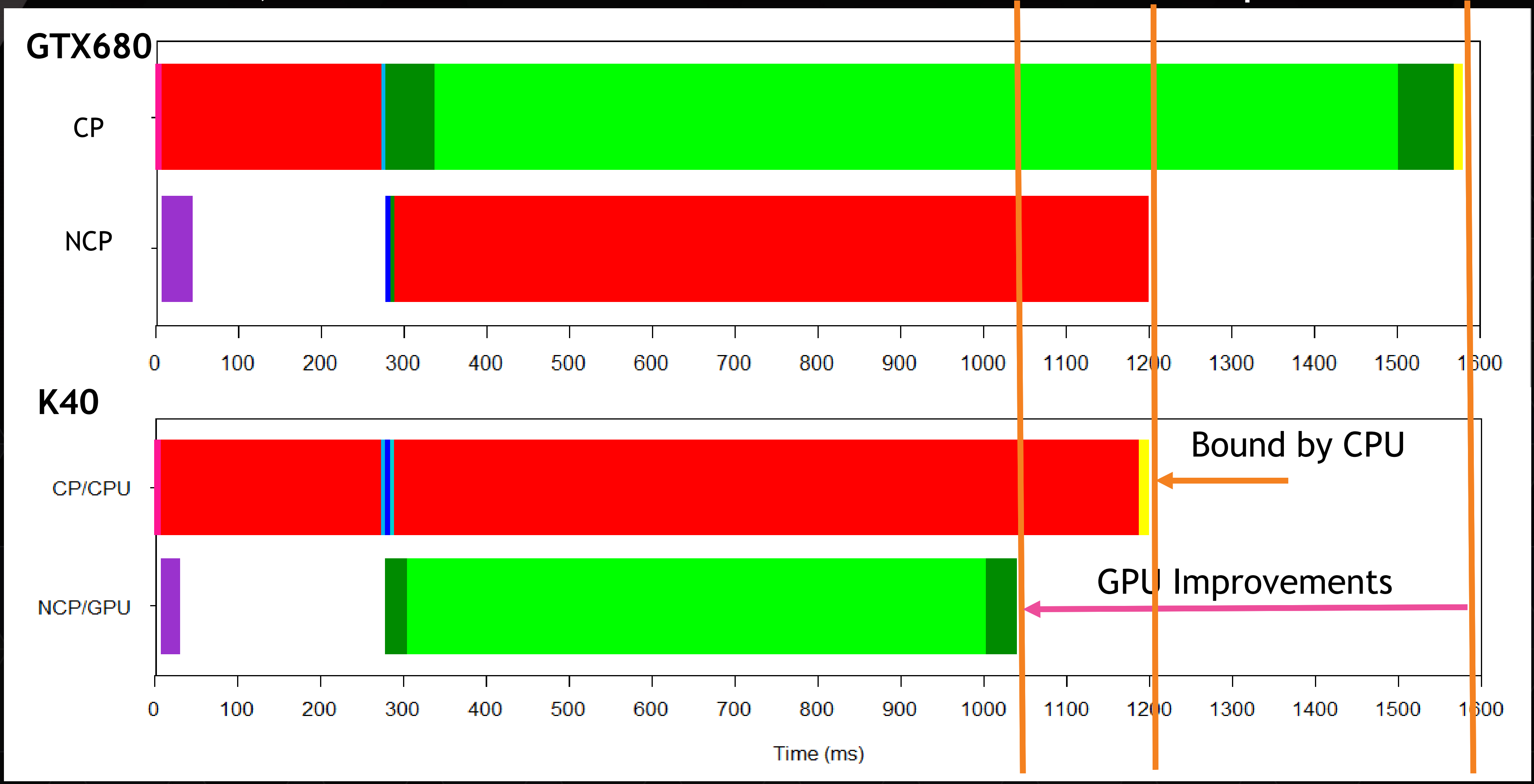
GPU TECHNOLOGY CONFERENCE

PREDICTION WITHOUT CRITICAL PATH INFO

Subtract the GPU improvements from total time, we got ~1.5X speedup.

API name	start	GTX680	K40X	where
cudaStreamSynchronize_CPU	0	0	0	CPU
cudaMemsetD8Async_CPU	0	7	7	CPU
cudaMemsetD8Async_GPU	7	20	12	GPU
CPUWork_CPU	7	266	266	CPU
cudaMemcpyHtoDAsync_CPU	273	4	4	CPU
cudaMemcpyHtoDAsync_GPU	277	60	27	GPU
cudaLaunchKernel_CPU	277	7	7	CPU
cudaMemcpyDtoHAsync_CPU	284	4	4	CPU
CPUWork_CPU	288	900	900	CPU
nbnxn_kernel_ElecEwQSTab_VdwLJ_F_cuda	337	1165	699	GPU
cudaMemcpyDtoHAsync_GPU	1502	66	26	GPU
cudaStreamSynchronize_CPU	1568	0	0	CPU
Total Time	1568		1066	

But, this is NOT exact if we don't consider critical path!



The critical path changed after we use faster GPU, and than CPU becomes bottleneck.

	Tested on GTX 680				Predict for K40X			
API name	start	duration	where	type	start	duration	where	type
cuStreamSynchronize_CPU	0	0	CPU	CP	0	0	CPU	CP
cuMemsetD8Async_CPU	0	7	CPU	CP	0	7	CPU	CP
cuMemsetD8Async_GPU	7	20	GPU	NCP	7	12	GPU	NCP
CPUWork_CPU	7	266	CPU	CP	7	266	CPU	CP
cuMemcpyHtoDAsync_CPU	273	4	CPU	CP	273	4	CPU	CP
cuMemcpyHtoDAsync_GPU	277	60	GPU	CP	277	27	GPU	NCP
cuLaunchKernel_CPU	277	7	CPU	NCP	277	7	CPU	CP
cuMemcpyDtoHAsync_CPU	284	4	CPU	NCP	284	4	CPU	CP
CPUWork_CPU	288	900	CPU	NCP	288	900	CPU	CP
nbnxn_kernel_ElecEwQSTab_VdwLJ_F_cuda	337	1165	GPU	CP	304	699	GPU	NCP
cuMemcpyDtoHAsync_GPU	1502	66	GPU	CP	1003	26	GPU	NCP
cuStreamSynchronize_CPU	**1568**	0	CPU	CP	**1188**	0	CPU	CP

GPU TECHNOLOGY CONFERENCE

THANK YOU

JOIN THE CONVERSATION

#GTC15