Название курса

Краткое описание

Программа

Критерии оценивания

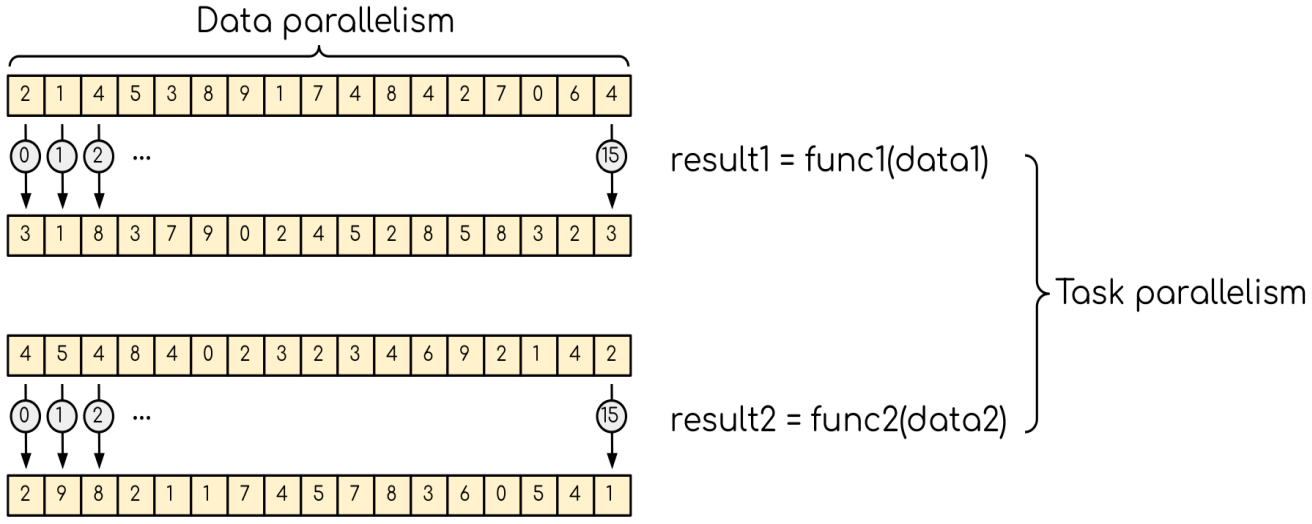
Полезные ссылки

Введение

Асинхронное выполнение

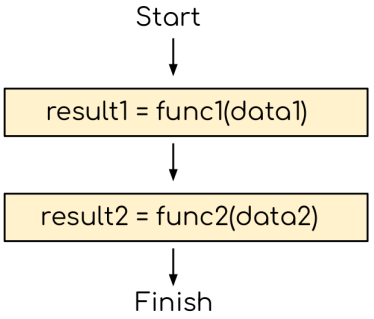
1. Параллелизм по задачам

So far when using GPUs we have explored the data parallelism. In many applications there is another type of parallelism, called task parallelism. A good example is molecular dynamics, where particle exert forces of different nature. These include short-range and long-range electrostatic interactions, bonded forces. Since these forces can be computed independently, their evaluation can run in parallel. It worth noting that data parallelism can be seen as the task parallelism. Indeed, if we do operation on a vector, this vector can be split in two parts and each part can be considered as a separate task. For example, this can be explored in the reduction example — different part of the reduction array can be reduced independently. All that we will have to do after is to sum the results up. Note that there is another avenue that can be explored in a GPU version as we will see later.



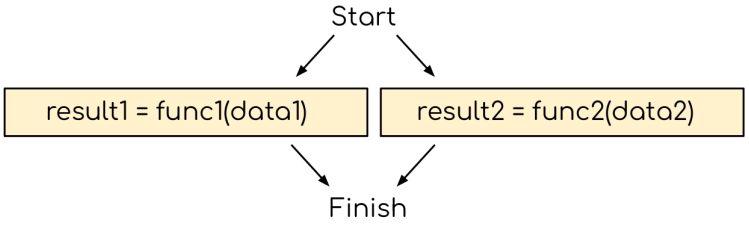
Data parallelism and task parallelism. The data parallelism is when the same operation applies to multiple data (e.g. multiple elements of an array are transformed). The task parrallelism implies that there are more than one independent task that, in principle, can be executed in parallel.

A typical program running on a single CPU core will be executed sequentially, as shown on the figure below.



The CPU program that does not explore the task based parallelism. The two functions can be issued independently, but still executed sequentially.

With more than on CPU code, one can start exploring the task-based parallelism, running the two independent task simultaneously (e.g. one task per core or MPI rank). This is out of scope of this workshop, but we encourage listeners to participate in our `MPI training <https://enccs.github.io/intermediate-mpi/>`_.).



Exploring the task parallelism. Since two functions are independent, they can be executed in parallel (e.g. on two CPU cores or in two MPI ranks).

To see how the task parallelism can be explored, let us consider the following example. We have two vectors of values from 0 to 1. Let us apply moderately compute-intensive function to each element of each vectors. In this example we will use the folded trigonometric functions. Now we have both data and task parallelism in the same example: we do the same operation on multiple vector elements and we have two vectors to execute it on. Let us start with exploring the data parallelism first.

⌨ Task parallelism

C++ code | Solution

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <math.h>
#include <time.h>

static constexpr int numIterations = 100;
static constexpr int numValuesToPrint = 10;

void func1(const float* in, float* out, int numElements)
{
    for (int i = 0; i < numElements; i++)
    {
        float value = in[i];
        for (int iter = 0; iter < numIterations; iter++)
        {
            value = std::sin(value);
        }
        out[i] = value;
    }
}

void func2(const float* in, float* out, int numElements)
{
    for (int i = 0; i < numElements; i++)
    {
        float value = in[i];
        for (int iter = 0; iter < numIterations; iter++)
        {
            value = -std::sin(value);
        }
        out[i] = value;
    }
}

int main(int argc, char* argv[])
{

    int numElements = (argc > 1) ? atoi(argv[1]) : 1000000;

    printf("Transforming %d values.\n", numElements);

    float* data1   = (float*)calloc(numElements, sizeof(float));
    float* data2   = (float*)calloc(numElements, sizeof(float));

    srand(1214134);
    for (int i = 0; i < numElements; i++)
    {
        data1[i] = float(rand())/float(RAND_MAX + 1.0);
        data2[i] = float(rand())/float(RAND_MAX + 1.0);
    }

    // Timing
    clock_t start = clock();

    func1(data1, data1, numElements);
    func2(data2, data2, numElements);

    // Timing
    clock_t finish = clock();

    printf("The results are:\n");
```

```
    for (int i = 0; i < numValuesToPrint; i++)
    {
        printf("%f, %f\n", data1[i], data2[i]);
    }
    printf("...\n");
    for (int i = numElements - numValuesToPrint; i < numElements; i++)
    {
        printf("%f, %f\n", data1[i], data2[i]);
    }
    double sum1 = 0.0;
    double sum2 = 0.0;
    for (int i = 0; i < numElements; i++)
    {
        sum1 += data1[i];
        sum2 += data2[i];
    }
    printf("The summs are: %f and %f\n", sum1, sum2);

    printf("It took %f seconds\n", (double)(finish - start) / CLOCKS_PER_SEC);

    // Release the memory
    free(data1);
    free(data2);

    return 0;
}
```

Convert the CPU code to GPU code.
To compile the CPU code, type ``gcc async_cpu.cpp -lm -o async_cpu``.
Note that we are using math library, hence we need to tell the compiler to link against it.

1. Change the extension of the file to ``.cu`` so that ``nvcc`` understands that there will be CUDA code in the file.

2. Add ``h_`` prefixes to the CPU arrays, to avoid the confusion between host and device data.

3. Create and allocate memory for the GPU counterparts of the CPU data.
   Since we do not need to save the initial data in this example, the results may overwrite the input.

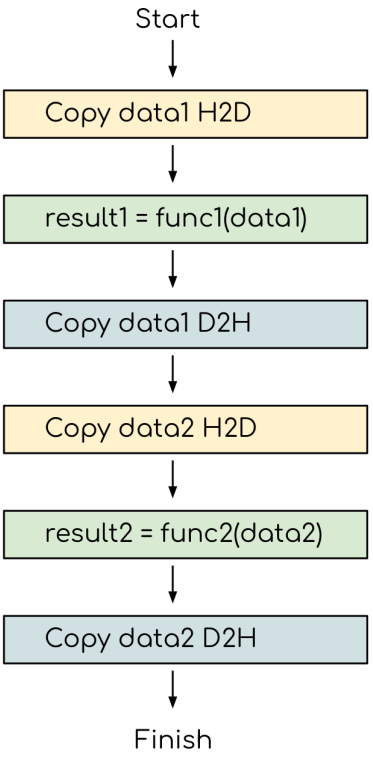
4. Copy the initial data to the GPU before running computing functions and copy the to-be results after.
   In this example, we will time the date transfers, so place them inside the timing region of the code.

5. Convert the compute functions to the GPU kernels: add |__global__| specifier and change loop index into the thread index.
   Make sure that you do not go out of bonds with a conditional.

6. Add kernel launch configuration to GPU kernels.
   Make sure that you pass the device pointers, not host pointers to the kernels.

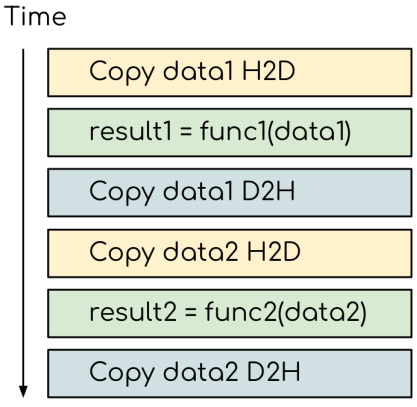
7. Compile the code with ``nvcc async_gpu_1.cu -o async_gpu_1``.
   Execute the program and make sure that the results are the same as in the CPU version.

Although the evaluation of these fairly compute heavy kernels is noticeable quicker on a GPU, we still have the room for improvement. The current version of the program requests GPU to do the following workflow:



The GPU program that does not explore the task based parallelism. All the data transfers and two functions are executed sequentially.

As a result, the execution timeline looks similar to this:

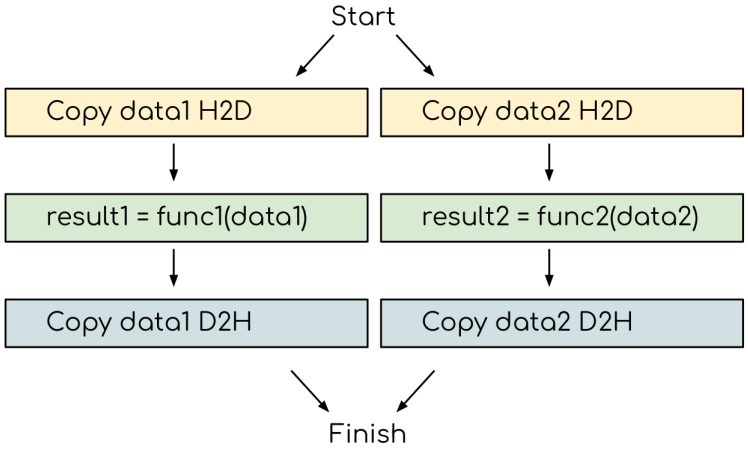


The execution timeline of the sequential GPU program. The order of the blocks may shuffle depending on the order you have issued the copy calls and kernels, but the total time will still be the sum of execution time for all the tasks.

2. Exploring task parallelism

A GPU-accelerated part of a typical program stars with the data transfer from host to the device. Then the computational kernel is called and the data is copied back. Consider now having two GPU tasks that need their own data. Both of these task will copy the input host

to the device, execute the computational kernel and copy data back. So we want the execution to follow the scheme below.



The GPU program that does not explore the task based parallelism. All the data transfers and two functions are executed sequentially.

Note that there are still dependencies between tasks: we can not run the `func1(..)` before the `data1` is on the GPU and we can not copy the `result1` to the CPU before the kernel is finished. In order to express such sequential dependencies implicitly in CUDA, there is a mechanism called streams. Essentially, you assign the asynchronous API calls to a stream, where they ensured to execute one after another. Different streams, on the other hand, can run simultaneously. Creating a stream is done by calling the following function:

`cudaStreamCreate`

```
__host__ cudaError_t cudaStreamCreate(cudaStream_t* stream)
```

This function can only be called from the host code and will return |cudaError_t| object if something went wrong. It takes a pointer to a |cudaStream_t| object, which should be initialized. The later can be constructed by:

```
cudaStream_t stream;
```

In order to use the streams for GPU kernel execution, one has to modify the kernel launch config by adding stream as a forth argument:

```
gpu_kernel<<<numBlocks, threadsPerBlock, sharedMemorySizeInBytes, stream>>>(..)
```

As we saw in the previous example, the third argument is the size of the shared memory needed by the kernel. We do not need it in this example, but this is alright, since we can always request 0 bytes. Calling GPU kernel in a stream results in its asynchronous launch. Now we need to make the data transfer asynchronous to. Before we do that, we need to do some preparations with our data. Any asynchronous call returns the execution back to the CPU, so we need to ensure that the host data is not tempered with while it is copied to the GPU. This is also called pinning, and should be done by using CUDA API while allocating host memory:

cudaMallocHost

```
    __host__ cudaError_t cudaMallocHost(void** ptr, size_t size)
```

The function works the same way as the |cudaMalloc|, we are already familiar with. It takes the pointer to the address in memory where allocation should happen and size of the allocation in bytes. Note that trying to release this memory with usual `free(...)` call will result in segmentation fault error. To release the pinned memory, one should use the CUDA API function.

cudaFreeHost

```
    __host__ cudaError_t cudaFreeHost(void* ptr)
```

Now the host arrays are pinned, we can do the host to device and device to host copies asynchroneously.

cudaMemcpyAsync

```
    __host__ __device__ cudaError_t cudaMemcpyAsync(void* dst, const void* src,
size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0)
```

The signature of this function is very similar to the synchronous variant we used before. The only difference is that it now takes one extra argument — the stream in which the copy should be executed.

One last function that we are going to need is the blocking synchronization function. The calls to asynchronous functions return control back to the CPU right after the call, before the actual execution is completed. Hence, before the data is analyzed back on the CPU, we

need to ensure that all the calls we issued are completed. This can be done with the following function from CUDA API:

cudaDeviceSynchronize

```
__host__ __device__ cudaError_t cudaDeviceSynchronize()
```

Now we have all the means to execute the data transfers and kernel calls asynchronously.

Asynchronous code

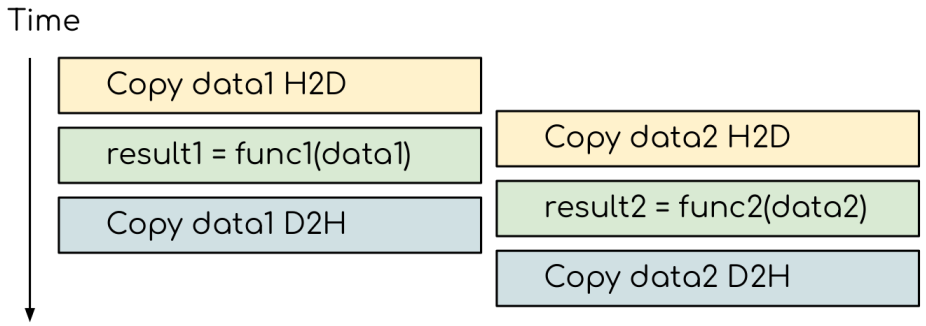
Synchronos code | Asynchronos code

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <time.h>

#define BLOCK_SIZE 256

static constexpr int numIterations = 100;
static constexpr int numValuesToPrint = 10;

__global__ void func1_kernel(const float* in, float* out, int numElements)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i < numElements)
    {
        float value = in[i];
        for (int iter = 0; iter < numIterations; iter++)
        {
            value = sinf(value);
        }
        out[i] = value;
    }
}

__global__ void func2_kernel(const float* in, float* out, int numElements)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i < numElements)
    {
        float value = in[i];
        for (int iter = 0; iter < numIterations; iter++)
        {
            value = -sinf(value);
        }
        out[i] = value;
    }
}

int main(int argc, char* argv[])
{

    int numElements = (argc > 1) ? atoi(argv[1]) : 1000000;

    printf("Transforming %d values.\n", numElements);

    float* h_data1   = (float*)calloc(numElements, sizeof(float));
    float* h_data2   = (float*)calloc(numElements, sizeof(float));

    srand(1214134);
    for (int i = 0; i < numElements; i++)
    {
        h_data1[i] = float(rand())/float(RAND_MAX + 1.0);
        h_data2[i] = float(rand())/float(RAND_MAX + 1.0);
    }

    int threadsPerBlock = BLOCK_SIZE;
    int numBlocks = numElements/BLOCK_SIZE + 1;

    float* d_data1;
    float* d_data2;

    cudaMalloc((void**)&d_data1, numElements*sizeof(float));
```

```
    cudaMalloc((void**)&d_data2, numElements*sizeof(float));

    // Timing
    clock_t start = clock();

    cudaMemcpy(d_data1, h_data1, numElements*sizeof(float),
cudaMemcpyHostToDevice);
    func1_kernel<<<numBlocks, threadsPerBlock>>>(d_data1, d_data1,
numElements);
    cudaMemcpy(h_data1, d_data1, numElements*sizeof(float),
cudaMemcpyDeviceToHost);

    cudaMemcpy(d_data2, h_data2, numElements*sizeof(float),
cudaMemcpyHostToDevice);
    func2_kernel<<<numBlocks, threadsPerBlock>>>(d_data2, d_data2,
numElements);
    cudaMemcpy(h_data2, d_data2, numElements*sizeof(float),
cudaMemcpyDeviceToHost);

    // Timing
    clock_t finish = clock();

    printf("The results are:\n");
    for (int i = 0; i < numValuesToPrint; i++)
    {
        printf("%f, %f\n", h_data1[i], h_data2[i]);
    }
    printf("...\n");
    for (int i = numElements - numValuesToPrint; i < numElements; i++)
    {
        printf("%f, %f\n", h_data1[i], h_data2[i]);
    }
    double sum1 = 0.0;
    double sum2 = 0.0;
    for (int i = 0; i < numElements; i++)
    {
        sum1 += h_data1[i];
        sum2 += h_data2[i];
    }
    printf("The summs are: %f and %f\n", sum1, sum2);

    printf("It took %f seconds\n", (double)(finish - start) / CLOCKS_PER_SEC);

    // Release the memory
    free(h_data1);
    free(h_data2);

    cudaFree(d_data1);
    cudaFree(d_data2);

    return 0;
}
```

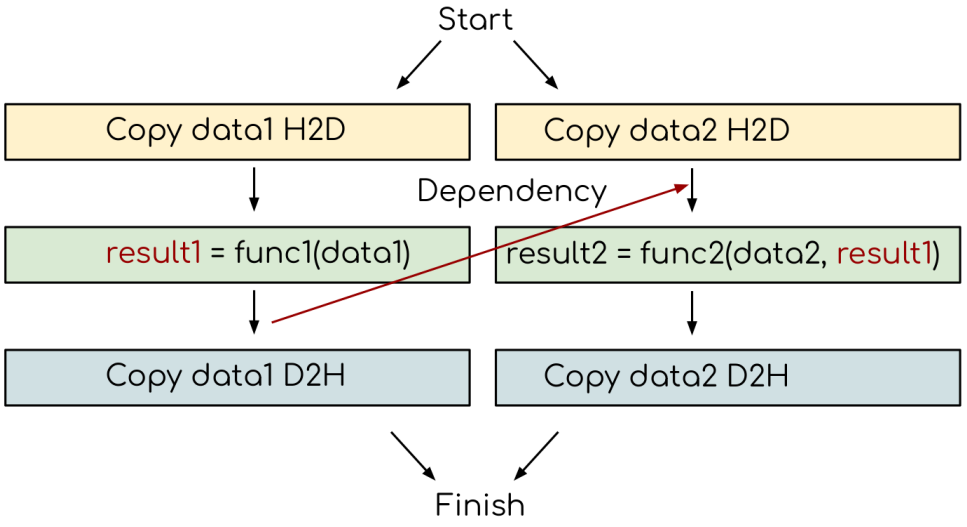
```
1. The CPU data needs to be pinned.
   Change the allocation and release of the memory so that CUDA API calls to
|cudaMallocHost| and |cudaFreeHost| are used.

2. Create two streams --- one for ``func1(..)`` and one for ``func2()``.

3. Change host to device and device to host data transfers from |cudaMemcpy| to
|cudaMemcpyAsync|.
   Use ``stream1`` for ``data1`` and ``stream2`` for ``data2``.

4. Make the kernel launch asynchronous by adding streams as forth parameter to the
kernel launch configurations.
   You will also need to specify the shared memory size as the third parameter ---
set it to zero.

5. Add |cudaDeviceSynchronize| call when all asynchronous API calls are made to
make sure that everything is finished before the results are printed.
```

On a GPU, the host to device copy, kernel evaluation and device to host copy require different resources. Hence, while the data is copied, GPU can execute the computational kernel without interfering with the compute.



The execution timeline of the asynchronous GPU program. GPU runtime will overlap the submitted tasks to an extent that they do not interfere with each other.

3. Introducing dependencies between kernels



Adding extra dependency between two tasks.

Let us now consider a case, where there is an extra dependency between tasks. Assume that the `func2(..)` not needs a result of the `func1(..)` to be evaluated. This is easy to da in the synchronous version of the program.

⌨ Code with dependencies

Syncronous code with independent kernels | C++ code

Syncronous code with dependencies between kernels

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <time.h>

#define BLOCK_SIZE 256

static constexpr int numIterations = 100;
static constexpr int numValuesToPrint = 10;

__global__ void func1_kernel(const float* in, float* out, int numElements)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i < numElements)
    {
        float value = in[i];
        for (int iter = 0; iter < numIterations; iter++)
        {
            value = sinf(value);
        }
        out[i] = value;
    }
}

__global__ void func2_kernel(const float* in, float* out, int numElements)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i < numElements)
    {
        float value = in[i];
        for (int iter = 0; iter < numIterations; iter++)
        {
            value = -sinf(value);
        }
        out[i] = value;
    }
}

int main(int argc, char* argv[])
{

    int numElements = (argc > 1) ? atoi(argv[1]) : 1000000;

    printf("Transforming %d values.\n", numElements);

    float* h_data1   = (float*)calloc(numElements, sizeof(float));
    float* h_data2   = (float*)calloc(numElements, sizeof(float));

    srand(1214134);
    for (int i = 0; i < numElements; i++)
    {
        h_data1[i] = float(rand())/float(RAND_MAX + 1.0);
        h_data2[i] = float(rand())/float(RAND_MAX + 1.0);
    }

    int threadsPerBlock = BLOCK_SIZE;
    int numBlocks = numElements/BLOCK_SIZE + 1;

    float* d_data1;
    float* d_data2;

    cudaMalloc((void**)&d_data1, numElements*sizeof(float));
```

```
    cudaMalloc((void**)&d_data2, numElements*sizeof(float));

    // Timing
    clock_t start = clock();

    cudaMemcpy(d_data1, h_data1, numElements*sizeof(float),
cudaMemcpyHostToDevice);
    func1_kernel<<<numBlocks, threadsPerBlock>>>(d_data1, d_data1,
numElements);
    cudaMemcpy(h_data1, d_data1, numElements*sizeof(float),
cudaMemcpyDeviceToHost);

    cudaMemcpy(d_data2, h_data2, numElements*sizeof(float),
cudaMemcpyHostToDevice);
    func2_kernel<<<numBlocks, threadsPerBlock>>>(d_data2, d_data2,
numElements);
    cudaMemcpy(h_data2, d_data2, numElements*sizeof(float),
cudaMemcpyDeviceToHost);

    // Timing
    clock_t finish = clock();

    printf("The results are:\n");
    for (int i = 0; i < numValuesToPrint; i++)
    {
        printf("%f, %f\n", h_data1[i], h_data2[i]);
    }
    printf("...\n");
    for (int i = numElements - numValuesToPrint; i < numElements; i++)
    {
        printf("%f, %f\n", h_data1[i], h_data2[i]);
    }
    double sum1 = 0.0;
    double sum2 = 0.0;
    for (int i = 0; i < numElements; i++)
    {
        sum1 += h_data1[i];
        sum2 += h_data2[i];
    }
    printf("The summs are: %f and %f\n", sum1, sum2);

    printf("It took %f seconds\n", (double)(finish - start) / CLOCKS_PER_SEC);

    // Release the memory
    free(h_data1);
    free(h_data2);

    cudaFree(d_data1);
    cudaFree(d_data2);

    return 0;
}
```

Create the synchronos version of the GPU code with dependency.
Use GPU version of the code without dependencies as a starting point and a CPU version of the code with dependencies as a reference.

4. Problem with running asynchronously

Let us now try to convert the code so it will run asynchronously as if there is no dependency between functions.

⌨ Problem with running asynchroneously

Syncronous code with independent kernels

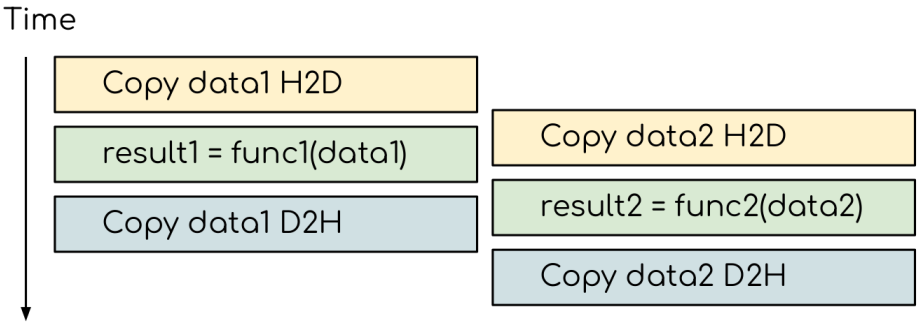
Syncronous code with dependencies between kernels

```
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <time.h>

#define BLOCK_SIZE 256

static constexpr int numIterations = 100;
static constexpr int numValuesToPrint = 10;

__global__ void func1_kernel(const float* in, float* out, int numElements)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i < numElements)
    {
        float value = in[i];
        for (int iter = 0; iter < numIterations; iter++)
        {
            value = sinf(value);
        }
        out[i] = value;
    }
}

__global__ void func2_kernel(const float* in1, const float* in2, float* out,
int numElements)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i < numElements)
    {
        float value1 = in1[numElements - i - 1];
        float value2 = in2[i];
        for (int iter = 0; iter < numIterations; iter++)
        {
            value2 = -sinf(value2);
        }
        out[i] = value1 + value2;
    }
}

int main(int argc, char* argv[])
{

    int numElements = (argc > 1) ? atoi(argv[1]) : 1000000;

    printf("Transforming %d values.\n", numElements);

    float* h_data1   = (float*)calloc(numElements, sizeof(float));
    float* h_data2   = (float*)calloc(numElements, sizeof(float));

    srand(1214134);
    for (int i = 0; i < numElements; i++)
    {
        h_data1[i] = float(rand())/float(RAND_MAX + 1.0);
        h_data2[i] = float(rand())/float(RAND_MAX + 1.0);
    }

    int threadsPerBlock = BLOCK_SIZE;
    int numBlocks = numElements/BLOCK_SIZE + 1;

    float* d_data1;
    float* d_data2;
```

```
    cudaMalloc((void**)&d_data1, numElements*sizeof(float));
    cudaMalloc((void**)&d_data2, numElements*sizeof(float));

    // Timing
    clock_t start = clock();

    cudaMemcpy(d_data1, h_data1, numElements*sizeof(float),
cudaMemcpyHostToDevice);
    func1_kernel<<<numBlocks, threadsPerBlock>>>(d_data1, d_data1,
numElements);
    cudaMemcpy(h_data1, d_data1, numElements*sizeof(float),
cudaMemcpyDeviceToHost);

    cudaMemcpy(d_data2, h_data2, numElements*sizeof(float),
cudaMemcpyHostToDevice);
    func2_kernel<<<numBlocks, threadsPerBlock>>>(d_data1, d_data2, d_data2,
numElements);
    cudaMemcpy(h_data2, d_data2, numElements*sizeof(float),
cudaMemcpyDeviceToHost);

    // Timing
    clock_t finish = clock();

    printf("The results are:\n");
    for (int i = 0; i < numValuesToPrint; i++)
    {
        printf("%f, %f\n", h_data1[i], h_data2[i]);
    }
    printf("...\n");
    for (int i = numElements - numValuesToPrint; i < numElements; i++)
    {
        printf("%f, %f\n", h_data1[i], h_data2[i]);
    }
    double sum1 = 0.0;
    double sum2 = 0.0;
    for (int i = 0; i < numElements; i++)
    {
        sum1 += h_data1[i];
        sum2 += h_data2[i];
    }
    printf("The summs are: %f and %f\n", sum1, sum2);

    printf("It took %f seconds\n", (double)(finish - start) / CLOCKS_PER_SEC);

    // Release the memory
    free(h_data1);
    free(h_data2);

    cudaFree(d_data1);
    cudaFree(d_data2);

    return 0;
}
```

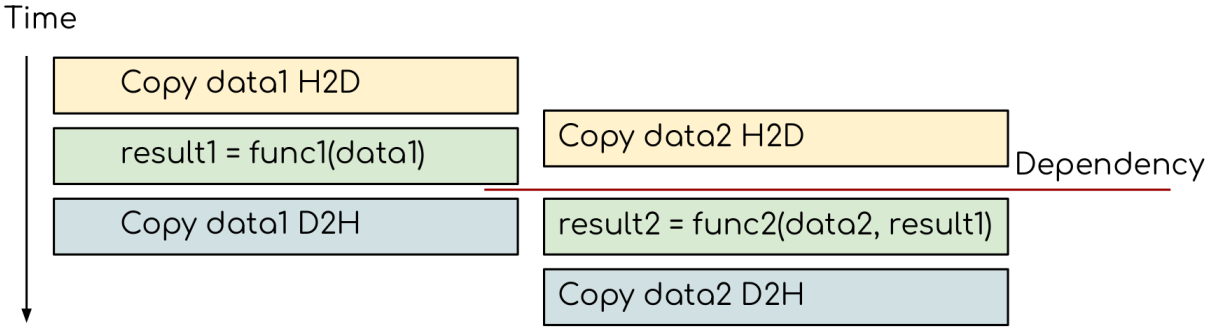
```
1. The CPU data needs to be pinned.
   Change the allocation and release of the memory so that CUDA API calls to
|cudaMallocHost| and |cudaFreeHost| are used.

2. Create two streams --- one for ``func1(..)`` and one for ``func2()``.

3. Change host to device and device to host data transfers from |cudaMemcpy| to
|cudaMemcpyAsync|.
   Use ``stream1`` for ``data1`` and ``stream2`` for ``data2``.

4. Make the kernel launch asynchronous by adding streams as forth parameter to the
kernel launch configurations.
   You will also need to specify the shared memory size as the third parameter ---
set it to zero.

5. Add |cudaDeviceSynchronize| call when all asynchronous API calls are made to
make sure that everything is finished before the results are printed.
```

As one can see, the resulting sum of the elements is now different. This happens because the evaluation of the `func1(..)` was not finished before the `func2(..)` call. So some values of the `result` were not updated for the `func2(..)`.



Adding extra dependency between two tasks.

5. Introducing dependencies between GPU tasks



Adding extra dependency between two tasks.

CUDA has a mechanism for introducing the explicit dependencies between streams, called events. First, one needs to create an |cudaEvent_t| object, which is done by the |cudaEventCreate| function.

`cudaEventCreate`

```
    __host__ cudaError_t cudaEventCreate(cudaEvent_t* event)
```

This function will initialize its only argument. The events can only be created on host, since one does not need one for each GPU thread. As most of the CUDA API function, the return object can indicate that there was an error with the call.

With event created, we need to be able to record and wait for it, which is done using following functions:

cudaEventRecord

```
    __host__ __device__ cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t
stream = 0)
```

This will record an event in the provided stream. Having the event recorded allows the host to see at which point the stream execution currently is. The function can be called from both host and device. The later can be useful for complex GPU kernel, where the event can be marked earlier than the kernel is complete. Note that if stream is not specified, the event will be marked in the default stream.

In order for another stream to wait until the event is recorded, one can use the followind function:

cudaStreamWaitEvent

```
    __host__ __device__ cudaError_t cudaStreamWaitEvent(cudaStream_t stream,
cudaEvent_t event, unsigned int flags = 0)
```

This takes an event and stream as arguments. Calling this function will stop the execution in the provided stream until the event is recorded. For this to be useful, the stream should be diferent to the one where event was recorded.

⌨ Introducing dependencies

Syncronous code with independent kernels

Syncronous code with dependencies between kernels

Syncronous code with dependencies between kernels

```
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <time.h>

#define BLOCK_SIZE 256

static constexpr int numIterations = 100;
static constexpr int numValuesToPrint = 10;

__global__ void func1_kernel(const float* in, float* out, int numElements)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i < numElements)
    {
        float value = in[i];
        for (int iter = 0; iter < numIterations; iter++)
        {
            value = sinf(value);
        }
        out[i] = value;
    }
}

__global__ void func2_kernel(const float* in1, const float* in2, float* out,
int numElements)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i < numElements)
    {
        float value1 = in1[numElements - i - 1];
        float value2 = in2[i];
        for (int iter = 0; iter < numIterations; iter++)
        {
            value2 = -sinf(value2);
        }
        out[i] = value1 + value2;
    }
}

int main(int argc, char* argv[])
{

    int numElements = (argc > 1) ? atoi(argv[1]) : 1000000;

    printf("Transforming %d values.\n", numElements);

    float* h_data1   = (float*)calloc(numElements, sizeof(float));
    float* h_data2   = (float*)calloc(numElements, sizeof(float));

    srand(1214134);
    for (int i = 0; i < numElements; i++)
    {
        h_data1[i] = float(rand())/float(RAND_MAX + 1.0);
        h_data2[i] = float(rand())/float(RAND_MAX + 1.0);
    }

    int threadsPerBlock = BLOCK_SIZE;
    int numBlocks = numElements/BLOCK_SIZE + 1;

    float* d_data1;
    float* d_data2;
```

```
    cudaMalloc((void**)&d_data1, numElements*sizeof(float));
    cudaMalloc((void**)&d_data2, numElements*sizeof(float));

    // Timing
    clock_t start = clock();

    cudaMemcpy(d_data1, h_data1, numElements*sizeof(float),
cudaMemcpyHostToDevice);
    func1_kernel<<<numBlocks, threadsPerBlock>>>(d_data1, d_data1,
numElements);
    cudaMemcpy(h_data1, d_data1, numElements*sizeof(float),
cudaMemcpyDeviceToHost);

    cudaMemcpy(d_data2, h_data2, numElements*sizeof(float),
cudaMemcpyHostToDevice);
    func2_kernel<<<numBlocks, threadsPerBlock>>>(d_data1, d_data2, d_data2,
numElements);
    cudaMemcpy(h_data2, d_data2, numElements*sizeof(float),
cudaMemcpyDeviceToHost);

    // Timing
    clock_t finish = clock();

    printf("The results are:\n");
    for (int i = 0; i < numValuesToPrint; i++)
    {
        printf("%f, %f\n", h_data1[i], h_data2[i]);
    }
    printf("...\n");
    for (int i = numElements - numValuesToPrint; i < numElements; i++)
    {
        printf("%f, %f\n", h_data1[i], h_data2[i]);
    }
    double sum1 = 0.0;
    double sum2 = 0.0;
    for (int i = 0; i < numElements; i++)
    {
        sum1 += h_data1[i];
        sum2 += h_data2[i];
    }
    printf("The summs are: %f and %f\n", sum1, sum2);

    printf("It took %f seconds\n", (double)(finish - start) / CLOCKS_PER_SEC);

    // Release the memory
    free(h_data1);
    free(h_data2);

    cudaFree(d_data1);
    cudaFree(d_data2);

    return 0;
}
```

```
The problem can be solved by adding an extra |cudaDeviceSynchronize| function call,
but this approach is not very flexible.
For instance, it will not work if there another independent stream in the program.
Preferred approach is to add explicit dependencies with events.

1. Create and initialize an event.

2. Record an event in ``stream1`` when ``func1(..)`` is evaluated.

3. Wait for an event in ``stream2`` before starting evaluating ``func2(..)``.
```

Inserting the dependency shoud fix the issue.

6. Further information

Setting up the dependencies so that the code performs at its best can be complicated. In CUDA, this can be done automatically with CUDA Graphs. CUDA Graphs allows one to record the sequence of API calls and merge them to improve the performance.

Also, do not forget that the CPU is also very capable device that should not be idling while GPU computes. Likely, asynchronous API calls make it straightforward to use the CPU: the execution is returned to the host immediately after the GPU call.