CHAPTER 18

Beef Up Flow Graphs with Async Nodes

Back in 2005, Herb Sutter wrote "The free lunch is over"[1] paper to warn us about the dawn of the multicore era and its implications on software development. In the multicore era, developers who care about performance can no longer sit back and lazily wait for the next processor generation in order to gleefully see their apps running faster. Those days are long gone. Herb's message was that developers who care about fully exploiting modern processors would have to embrace parallelism. At this point of the book, we certainly know this, so what? Well, we believe that today "Lunch is getting much too expensive." Let's elaborate on this.

In more recent years, strongly spurred by energy constraints, more complex processors have emerged. Nowadays, it is not difficult to find heterogeneous systems comprising one or more GPU, FPGA, or DSP alongside one or more multicore CPUs. Much in the same way that we embraced parallelism to get the most of all CPU cores, now it may also make sense to offload part of the computations to these accelerators. But hey, this is tough! Yep, it is! If sequential programming was once the "free lunch," heterogeneous parallel programming today is more like a feast at a three-star Michelin restaurant – we have to pay, but it is oh so good!

And does TBB help in saving some of the dinner price? Of course! How do you dare doubt it? In this and the next chapter of our book, we walk through the features recently incorporated to the TBB library in order to help make lunch affordable again –we show how to offload computation to asynchronous devices thereby embracing heterogeneous computing. In this chapter, we will pick up the TBB Flow Graph interface and reinforce it with a new type of node: the `async_node`. In the next chapter, we will go even further and put Flow Graph on OpenCL steroids.

[1] "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," Herb Sutter. www.gotw.ca/publications/concurrency-ddj.htm.

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_18

Async World Example

Let's start with the simplest example that uses an `async_node`. We will illustrate why this particular flow graph node is useful, and we will also present a more complex example that will be useful for the next chapter.

Since there is nothing simpler than a "Hello World" code snippet, we propose an "`Async World`" alternative based on the flow graph API that includes an `async_node` in the graph. If you have questions about flow graphs in TBB, you may wish to read through Chapter 3 for a solid background and use the "Flow Graph" section in Appendix B as a reference for the APIs. The flow graph that we build in this first example is depicted in Figure 18-1.

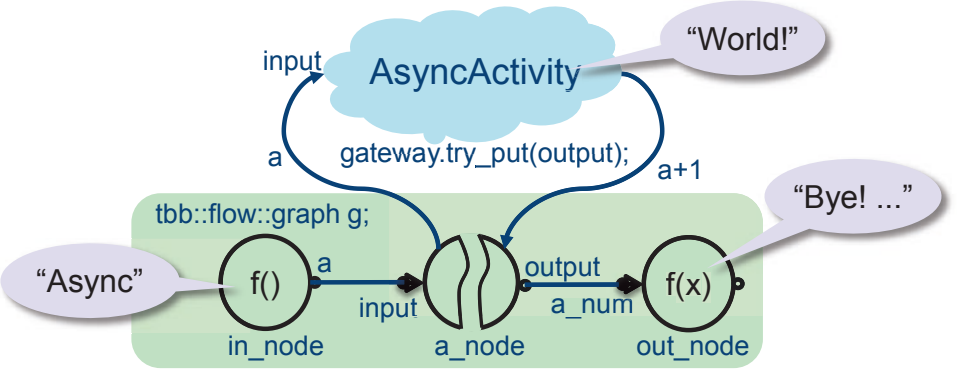

Figure 18-1. *Flow graph for the "Async world" example*

Our goal is to send a message from a `source_node`, `in_node`, to an `asynchronous node`, `a_node`, but instead of processing the message inside `a_node`, this task is offloaded to an asynchronous activity that is running somewhere (a different MPI node, an OpenCL capable GPU, an FPGA, you name it). As soon as this asynchronous task finishes, the flow graph engine has to get the control back, read the output of the async activity, and propagate the message to the descendent nodes in the graph. In our very simple "`Async World`" example, `in_node` just prints "`Async`" and passes `a=10` to `a_node`. The `a_node` receives `a=10` as the `input` and forwards it to an `AsyncActivity`. In this example, `AsyncActivity` is a class that just increments the input message and prints "`World!`". These two actions are carried out in a new thread that simulates here an asynchronous operation or device. Only when the `AsyncActivity` deigns to respond with `output=11`, `out_node` will receive this value and the program finishes.

The code presented in Figure 18-2 includes the `async_world()` function definition where we build the graph g composed of the three nodes of Figure 18-1.

```
void async_world() {
  tbb::flow::graph g;
  bool n = false;

  //Source node:
  tbb::flow::source_node<int> in_node{g,
    [&](int& a) {
      if (n) return false;
      std::cout << "Async ";
      a = 10;
      n = true;
      return true;
    },
    false
  };

  //Async node:
  AsyncActivity asyncAct;
  using activity_node_t = tbb::flow::async_node<int, int>;
  using gateway_t = activity_node_t::gateway_type;
  activity_node_t a_node{g, tbb::flow::unlimited,
    [&asyncAct](int const& input, gateway_t& gateway) {
      asyncAct.run(input, gateway);
    }
  };

  //Output node:
  tbb::flow::function_node<int> out_node{
    g, tbb::flow::unlimited,
    [](int const& a_num){
      std::cout << "Bye! Received: "<< a_num<< '\n';
    }
  };

  //Edges:
  make_edge(in_node, a_node);
  make_edge(a_node, out_node);

  //Run!
  in_node.activate();
  g.wait_for_all();
}
```

Figure 18-2. *Building the flow graph for the "`Async World`" example*

The source_node interface is described in the first entry of the table in Figure B-37 of Appendix B. In our example, we create the in_node of the source_node type. The argument of the lambda, int& a, is actually the output message that will be sent to its successor node in the graph (the async_node). When the source node is activated near the end of the async_world() function, by using in_node.activate(), the lambda will be executed just once because it returns "true" just for the first invocation (initially n=false, n is set to true inside the lambda, which only returns true if n=true). In this single invocation, a message with a=10 is sent to the next node in the graph. The last argument of the in_node is false so that the source node is created in hibernation mode and only wakes up after the in_node.activate() is called (otherwise the node start sending messages immediately after the output edge is connected).

Next comes the async_node definition. The syntax required for the async_node interface is

```
template<typename InType, typename OutType>
async_node(graph& g, size_t concurrency, Body body);
```

In our example, a_node is constructed here:

```
using activity_node_t = tbb::flow::async_node<int, int>;
using gateway_t = activity_node_t::gateway_type;
activity_node_t a_node{g, unlimited, []… /*lambda*/ };
```

which creates an async_node<int, int> in the graph g with unlimited concurrency. By using unlimited, we instruct the library to spawn a task as soon as a message arrives, regardless of how many other tasks have been spawned. Should we want only up to 4 concurrent invocations of a_node, we can change unlimited to 4. The template parameter <int, int> points out that a message of type int enters a_node and a message of type int leaves a_node. The lambda used in a_node constructor is the following:

```
[&asyncAct](int const& input, gateway_t& gateway) {
  asyncAct.run(input, gateway);
}
```

that captures by reference an AsyncActivity object, asyncAct, and declares the functor that has to be run for each message reaching a_node. This functor has two arguments, input and gateway, passed by reference. But wait, didn´t we say that the template parameter <int, int> means that the node expects an incoming integer and emits an

outgoing integer? Shouldn't the functor's prototype be `(const int& input) -> int`? Well, it would have been that way for a regular `function_node`, but we are dealing now with an `async_node` and its particularities. Here, we get `const int& input` as expected, but also a second input argument, `gateway_t& gateway`, that serves as an interface to inject the output of the `AsyncActivity` back into the graph. We will come to this trick when explaining the `AsyncActivity` class. For the moment, to finish the description of this node let's just say that it basically dispatches the `AsyncActivity` with `asyncAct.run(input, gateway)`.

The output node, `out_node`, is a `function_node` that in this example has been configured as an end node that does not send any output message:

```
tbb::flow::function_node<int> out_node{g, tbb::flow::unlimited,
  [](int const& a_num){
    std::cout << "Bye! Received: "<< a_num<< '\n';
  }
};
```

This node receives the integer that comes from the `AsyncActivity` through the `gateway` and finishes off just printing "`Bye!`" followed by the value of such integer.

In the last lines of our `Async World` example in Figure 18-2, we find two `make_edge` calls to create the connections depicted in Figure 18-1, and finally the graph is awakened with `in_node.activate()` to immediately wait until all messages have been processed with `g.wait_for_all()`.

Here comes the `AsyncActivity` class, which implements the asynchronous computations in our example as can be seen in Figure 18-3.

```
class AsyncActivity {
public:
  ~AsyncActivity() {
    asyncThread.join();
  }

  using node_t = tbb::flow::async_node<int, int>;
  using gateway_t = node_t::gateway_type;

  void run(int input, gateway_t& gateway) {
    gateway.reserve_wait();
    asyncThread = std::thread{
      [&,input]() {
        std::cout << "World! Input: " << input << '\n';
        int output = input + 1;
        gateway.try_put(output);
        gateway.release_wait();
      }
    };
  }
private:
   std::thread asyncThread;
};
```

Figure 18-3. *Implementation of the asynchronous activity*

The public member function "run" (that was invoked in a_node's functor with asyncAct.run) first does gateway.reserve_wait(), which notifies the flow graph that work has been submitted to an external activity so this can be taken into account by g.wait_for_all() at the end of async_world(). Then, an asynchronous thread is spawned to execute a lambda, which captures the gateway by reference and the input integer by value. It is key to pass input by value because otherwise the referenced variable, a in the source_node, can be destroyed before the thread reads its value (if the source_node finishes before the asyncThread can read the value of a).

The lambda in the thread constructor first prints the "World" message and then assigns output=11 (input+1, more precisely). This output is communicated back into the flow graph by calling the member function gateway.try_put(output). Finally, with gateway.release_wait(), we inform the flow graph that, as far as the AsyncActivity is concerned, there is no need to wait any longer for it.

Note There is no requirement to call member function `reserve_wait()` for each input message submitted to an external activity. The only requirement is that each call to `reserve_wait()` must have a corresponding call to `release_wait()`. Note that `wait_for_all()` will not exit while there are some `reserve_wait()` calls without matching `release_wait()`'s.

The output of the resulting code is

```
Async World! Input: 10
Bye! Received: 11
```

where "`Async`" is written by `in_node`, "`World! Input: 10`" by the asynchronous task and the last line by `out_node`.

Why and When async_node?

Now, there may be readers displaying a conceited smirk and thinking sort of "I don't need an `async_node` to implement this." Why don't we just rely on the good ol' `function_node`?

For example, a `a_node` could have been implemented as in Figure 18-4, where we use a `function_node` that receives an integer, `input`, and returns another integer, `output`. The corresponding lambda expression spawns a thread, `asyncThread`, that prints and generates the `output` value, and then waits for the thread to finish with the `asyncThread.join()` to gleefully return `output`.

```
tbb::flow::function_node<int, int> a_node{g,
  tbb::flow::unlimited,
  [&](const int& input) -> int {
    int output;
    std::thread asyncThread{[&,input]{
      std::cout << "World! Input: "<< input << '\n';
      output = input + 1;
    }};
   asyncThread.join(); // a worker thread blocks here!
   return output;
 }};
```

Figure 18-4. *A simplest implementation that creates and waits for an asynchronous thread. Did someone say DANGER?*

If you were not one of the smirking readers before, what about now? Because, what is wrong with this much simpler implementation? Why not rely on the same approach to also offload computations to a GPU or an FPGA, and wait for the accelerator to finish its duty?

To answer these questions, we have to bring back one of the fundamental TBB design criteria, namely the composability requirement. TBB is a composable library because performance does not take a hit if the developer decides or needs to nest parallel patterns inside other parallel patterns, no matter how many levels are nested. One of the factors that make TBB composable is that adding nested levels of parallelism does not increase the number of worker threads. That in turn avoids oversubscription and its associated overheads from ruining our performance. To make the most out of the hardware, TBB is usually configured so that it runs as many worker threads as logical cores. The various TBB algorithms (nested or not) only add enough user-level lightweight tasks to feed these worker threads and thereby exploit the cores. However, as we warned in Chapter 5, calling a blocking function inside a user-level task not only blocks the task but it also blocks the OS-managed worker thread processing this task. In such an unfortunate case, if we had a worker thread per core and one of them was blocked, the corresponding core may become idle. In such a case, we would not be fully utilizing the hardware!

In our simple example of Figure 18-4, the `asyncThread` will use the idle core when it runs the task outside the flow graph control. But what about offloading work to an accelerator (GPU/FPGA/DSP, pick as you please!), and waiting for it? If a TBB task calls blocking functions from OpenCL, CUDA, or Thrust code (to name a few), the TBB worker running this task will inevitably block.

Before `async_node` was available in the flow graph list of nodes, a possible, although not ideal, workaround was to oversubscribe the system with one extra thread. To accomplish this (as described in more detail in Chapter 11), we usually rely on the following lines:

```
int cores = tbb::task_scheduler_init::default_num_threads();
tbb::task_scheduler_init(cores+1);
```

This solution is still viable if we don't require a flow graph in our code and just want to offload work to an accelerator from, say, a `parallel_invoke` or one of the stages of a `parallel_pipeline`. The caveat here is that we should know that the extra thread is going to be blocked most of the time while waiting for the accelerator. However, the glitch with this workaround is that there will be periods of time in which the system is oversubscribed (before and after the offloading operation or even while the accelerator driver decides to block[2] the thread).

To avoid this issue, `async_node` comes to our rescue. When the `async_node` task (usually its lambda) finishes, the worker thread that was taking care of this task switches to work on other pending tasks of the flow graph. This way, the worker thread does not block leaving an idle core. What it is key to remember is that before the `async_node` task finishes, the flow graph should be warned of that an asynchronous task is in flight (using `gateway.reserve_wait()`), and just after the asynchronous task re-injects its result back into the flow graph (with `try_put()`) we should notify that the asynchronous task has finished with `gateway.release_wait()`. Still smirking? If so, please tell us why.

A More Realistic Example

The triad function of the well-known STREAM benchmark[3] is a basic array operation, also called "linked triad," that essentially computes `C = A + α*B`, where `A`, `B`, and `C` are 1D arrays. It is therefore quite similar to the BLAS 1 `saxpy` operation that implements `A=A+α*B`, but writing the result in a different vector. Pictorially, Figure 18-5 helps in understanding this operation.

[2]When a thread offloads a kernel to the GPU using a blocking call the driver may not immediately block the calling thread. For example, some GPU drivers keep the thread spinning so that it will respond earlier to lightweight kernels, and finally block the thread after some time to avoid consuming resources while heavyweight kernels finish.

[3]John McCalpin, STREAM benchmark, `www.cs.virginia.edu/stream/ref.html`.

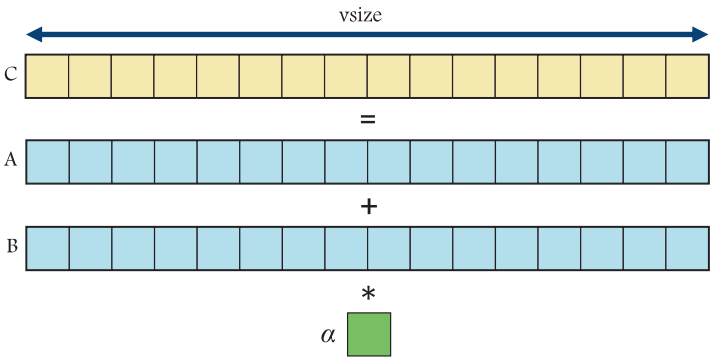

Figure 18-5. *Triad vector operation that computes* $C = A + \alpha * B$ $(c_i = a_i + \alpha * b_i, \forall i)$

In our implementation, we will assume that array sizes are determined by the variable `vsize` and that the three arrays store single precision floats. Coming up with a parallel implementation of this embarrassingly parallel algorithm is not challenging enough for us at this point of the book. Let's go for a heterogeneous implementation.

Okay, so you've got an integrated GPU? That don't impress me much![4] Reportedly, more than 95% of shipped processors come with an integrated GPU, sharing the die along with the multicore CPU. Would you sleep soundly after running the triad code on just one CPU core? Not quite, right? CPU cores shouldn't be sitting idle. Much in the same way, GPU cores shouldn't be sitting idle either. On many occasions, we can leverage the excellent GPU computing capabilities to further speed up some of our applications.

In Figure 18-6, we illustrate the way in which the triad computation will be distributed among the different computing devices.

[4]Shania Twain – That Don't Impress Me Much, *Come On Over* album, 1997.

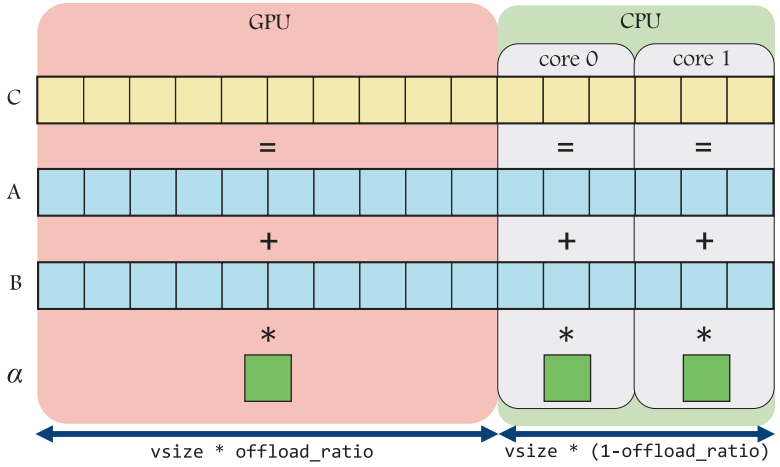

Figure 18-6. *Heterogeneous implementation of the triad computation*

In our implementation, we will rely on the `offload_ratio` variable, which controls the fraction of the iteration space that is offloaded to the GPU, whereas the rest is processed in parallel on the CPU. It goes without saying that $0 \leq$ `offload_ratio` ≤ 1.

The code will be based on the flow graph depicted in Figure 18-7. The first node, `in_node`, is a `source_node` that sends the same `offload_ratio` to `a_node` and `cpu_node`. The former is an `async_node` that offloads the computation of the corresponding subregion of the arrays to an OpenCL capable GPU. The latter is a regular `function_node` that nests a TBB `parallel_for` to split the CPU assigned subregion of the arrays among the available CPU cores. Execution time on both the GPU, `Gtime`, and CPU, `Ctime`, are collected in the corresponding node and converted into a tuple inside the `join_node`. Finally, in the `out_node`, those times are printed, and the heterogeneously computed version of array C is compared with a golden version obtained by a plain serial execution of the triad loop.

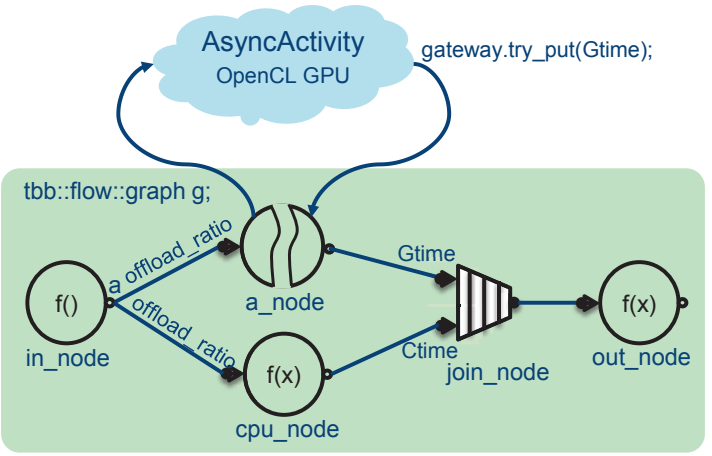

Figure 18-7. *Flow graph that implements heterogeneous triad*

Note We love to gently introduce new concepts, and we try to toe the line on this, especially when it comes to TBB content. However, OpenCL is outside the scope of the book, so we have to give up our own rule and just comment briefly on the OpenCL constructs that are used in the following examples.

For the sake of simplicity, in this example, we will accept the following assumptions:

1. In order to leverage a zero-copy buffer strategy that reduces the overhead of data movement between devices, we assume that an OpenCL 1.2 driver is available and that there is a common region of memory visible from both the CPU and the GPU. This is usually the case for integrated GPUs. For recent heterogeneous chips, OpenCL 2.0 is also available, and in such a case we can take advantage of the SVM (Shared Virtual Memory) as we will also illustrate next.

2. In order to reduce the number of arguments of the flow graph nodes and that way improve the readability of the code, pointers to CPU and GPU views of the three arrays `A`, `B`, and `C` are globally visible. The variable `vsize` is also global.

3. In order to skip the aspects less related to TBB, all the OpenCL boilerplate has been encapsulated into a single function `opencl_initialize()`. This function takes care of getting the platform, `platform`, selecting the GPU device, `device`, creating the GPU context, `context`, and a command queue, `queue`, reading the source code of the OpenCL kernel, compiling it to create the kernel, and initializing the three buffers that store the GPU view of arrays `A`, `B`, and `C`. Since the `AsyncActivity` also needs the command queue and the program handlers, the corresponding variables, `queue` and `program`, are also global. We took advantage of the C++ wrappers available for the OpenCL C API. More precisely, we used the `cl2.hpp` OpenCL C++ header that can be found on https://github.com/KhronosGroup/OpenCL-CLHPP/.

Let's start with the main function of the code; in Figure 18-8, we only show the definition of the two first nodes: `in_node` and `cpu_node`.

```
int main(int argc, const char* argv[]) {
  int nth = (argc>1) ? atoi(argv[1]): 4;
  vsize = (argc>2) ? atoi(argv[2]) : 100000000;
  float ratio = (argc>3) ? atof(argv[3]) : 0.5;
  float alpha = 0.5;
  opencl_initialize(); // OpenCL boilerplate
  tbb::task_scheduler_init init{nth};
  auto mp=tbb::global_control::max_allowed_parallelism;
  tbb::global_control gc(mp, nth+1); //One more thread, but sleeping
  tbb::flow::graph g;

  bool n = false;
  tbb::flow::source_node<float> in_node{g,
    [&](float& offload_ratio) -> bool {
    if(n) return false;
    offload_ratio = ratio;
    n = true;
    return true;
  },false};
```

Figure 18-8. *Main function of the heterogeneous triad computation with the first two nodes*

```
tbb::flow::function_node<float, double> cpu_node{g,
  tbb::flow::unlimited,
  [&](float offload_ratio) -> double {
    auto t=tbb::tick_count::now();
    tbb::parallel_for(
      tbb::blocked_range<size_t>{
        static_cast<size_t>(ceil(vsize*offload_ratio)),
        static_cast<size_t>(vsize)
      },
      [&](const tbb::blocked_range<size_t>& r){
        for (size_t i = r.begin(); i < r.end(); ++i)
          Chost[i] = Ahost[i] + alpha * Bhost[i];
      }
    );
    return (tbb::tick_count::now() - t).seconds();
  }};
// … to be continued …
```

Figure 18-8. (*continued*)

We first read the program arguments and initialize the OpenCL boilerplate calling `opencl_initialize()`. From this function, we only need to know that it initializes a GPU command queue, `queue`, and an OpenCL program, `program`. The initialization of the number of threads and the reason to initialize a `global_control` object will be addressed at the end of the section. The source code of the GPU kernel is quite straightforward:

```
kernel void triad(global float* A,
                  global float* B,
                  global float* C) {
  int i=get_global_id(0);
  float alpha = 0.5;
  C[i] = A[i] + alpha * B[i];
}
```

This implements the triad operation, `C = A + α*B`, assuming `α=0.5` and that the arrays of float are stored in global memory. At kernel launching time, we have to specify the range of iterations that the GPU will traverse, and the GPU internal scheduler will pick single iterations from this space with the instruction `i=get_global_id(0)`. For each one of these `i`'s, the computation `C[i] = A[i] + alpha * B[i]` will be conducted in parallel in the different compute units of the GPU.

Inside the `opencl_initialize()` function, we also allocate the three OpenCL buffers and the corresponding CPU pointers that point to the same buffers from the CPU side (what we call the CPU view of the arrays). Assuming we have OpenCL 1.2, for the input array A, we rely on the OpenCL `cl::Buffer` constructor to allocate a GPU accessible array called `Adevice`:

```
cl::Buffer Adevice;                    // Device view of A
Adevice = cl::Buffer{context, CL_MEM_READ_WRITE |
                     CL_MEM_ALLOC_HOST_PTR, sizeof(float)*vsize};
```

The flag `CL_MEM_ALLOC_HOST_PTR` is key to take advantage of the zero-copy buffer OpenCL feature because it forces the allocation of host-accessible memory. The same call is used for the other two GPU views of the arrays, `Bdevice` and `Cdevice`. To get a pointer to the CPU view of these buffers, the OpenCL `enqueueMapBuffer` is available and used as follows:

```
float* Ahost;                          // Host view of A
Ahost=(float*)queue.enqueueMapBuffer(Adevice, CL_TRUE, CL_MAP_WRITE |
CL_MAP_READ, 0, sizeof(float) * vsize, NULL, NULL, &err);
```

which gives us a float pointer `Ahost` that can be used from the CPU to read and write in the same memory region. Similar calls are needed for pointers `Bhost` and `Chost`. In modern processors with integrated GPUs, this call does not imply data copy overheads and hence the zero-copy buffer name for this strategy. There are additional subtleties regarding OpenCL like the meaning and functionality of `clEnqueueUnmapMemObject()` and potential issues arising of having both the CPU and GPU writing in different regions of the same array, but they fall beyond the scope of this book.

Note If your device supports OpenCL 2.0, the implementation is easier especially if the heterogeneous chip implements what is called fine-grained buffer SVM. In that case, it is possible to allocate a region of memory that not only is visible by both the CPU and GPU, but that can also be updated concurrently and kept coherent by the underlying hardware. In order to check whether or not OpenCL 2.0 and fine-grained buffer SVM are available, we need to use: `device.getInfo<CL_DEVICE_SVM_CAPABILITIES>();`

To exploit this feature, in the `opencl_initialize()`, we can use `cl::SVMAllocator()` and pass it as the allocator template argument of the `std::vector` constructor. This will give us a `std::vector A`, that is at the same time the GPU view and the CPU view of the data:

```
using svmalloc_t = cl::SVMAllocator<float,
                cl::SVMTraitFine<cl::SVMTraitReadWrite<>>>;
svmalloc_t svmAlloc;
A=std::vector<float, svmalloc_t>{vsize, 0, svmAlloc};
```

This is, no need for `Ahost` and `Adevice` anymore. Just `A`. As with any shared data, we are responsible of avoiding data races. In our example, this is easy because the GPU writes in a region of the array `C` that does not overlap with the region written by the CPU. If this condition is not met, in some cases, the solution is to resort to an array of atomics. Such a solution is usually called platform atomics, or system atomics, since they can be atomically updated by the CPU and the GPU. This feature is optionally implemented and requires that we instantiate the `SVMAllocator` with `cl::SVMTraitAtomic<>`.

The next thing in Figure 18-8 is the declaration of the graph `g` and definition of the `source_node`, `in_node`, which is quite similar to the one explained in Figure 18-2, with the single difference that it passes a message with the value of `offload_ratio`.

The next node in our example is a `function_node`, `cpu_node`, which receives a `float` (actually, `offload_ratio`) and sends a `double` (the time required to do the CPU computation). Inside the `cpu_node` lambda, a `parallel_for` is invoked and its first argument is a blocked range like this:

```
tbb::blocked_range<size_t>{
                     static_cast<size_t>(ceil(vsize*offload_ratio)),
                     static_cast<size_t>(vsize)
                    }
```

which means that only the upper part of the arrays will be traversed. The lambda of this `parallel_for` computes in parallel `Chost[i] = Ahost[i] + alpha * Bhost[i]` for different chunks of iterations in which the range is automatically partitioned.

We can continue in Figure 18-9 with the next node, a_node, that is an asynchronous node that receives a float (again the offload_ratio value) and sends the time required by the GPU computation. This is accomplished asynchronously in a_node's lambda where the member function run of an AsyncActivity object, asyncAct, is called, similarly to what we already saw in Figure 18-2.

```
using async_node_t = tbb::flow::async_node<float, double>;
using gateway_t = async_node_t::gateway_type;
AsyncActivity asyncAct;
async_node_t a_node{g, tbb::flow::unlimited,
  [&asyncAct](const float& offload_ratio, gateway_t& gateway) {
    asyncAct.run(offload_ratio, gateway);
  }
};
using join_t = tbb::flow::join_node <std::tuple<double,double>,
                                     tbb::flow::queueing>;
join_t node_join{g};

tbb::flow::function_node<join_t::output_type> out_node{g,
  tbb::flow::unlimited,
  [&](const join_t::output_type& times){
    std::vector<float> CGold(vsize);
    std::transform(Ahost, Ahost + vsize, Bhost, CGold.begin(),
                   [&](float a, float b)->float{return a+alpha*b;});
    // Compare golden triad with heterogeneous triad
    if (!std::equal(Chost, Chost+vsize, CGold.begin()))
       std::cout << "Error!!\n";
    std::cout<< "Time cpu: "<< std::get<1>(times)<< " sec."<< '\n';
    std::cout<< "Time gpu: "<< std::get<0>(times)<< " sec."<< '\n';
}};
// … to be continued …
```

Figure 18-9. *Main function of the heterogeneous triad computation with the definition of the last three nodes*

The join_node does not deserve our time here because it was already covered in Chapter 3. Suffice to say that it forwards a tuple, which packs the GPU time and the CPU time, to the next node.

The final node is a function_node, out_node, which receives the tuple with the times. Before printing them, it checks that the resulting C array has been correctly computed partially on the CPU and partially on the GPU. To this end, a golden version of C, CGold, is allocated and then computed serially using the STL algorithm transform. Then, if Chost coincides with CGold, we are all set. The equal STL algorithm comes in handy to implement this comparison.

Figure 18-10 finishes off the `main()` function with the node connections, thanks to five `make_edge` calls, followed by the `in_node` activation to trigger the execution of the graph. We wait for completion with `g.wait_for_all()`.

```
    tbb::flow::make_edge(in_node, a_node);
    tbb::flow::make_edge(in_node, cpu_node);
    tbb::flow::make_edge(a_node, tbb::flow::input_port<0>(node_join));
    tbb::flow::make_edge(cpu_node,tbb::flow::input_port<1>(node_join));
    tbb::flow::make_edge(node_join, out_node);

    auto gt = tbb::tick_count::now();
    in_node.activate();
    g.wait_for_all();
    return 0;
} // End of main()!
```

Figure 18-10. *Last part of the triad main function where nodes are connected and the graph is dispatched*

Finally, in Figure 18-11, we present the implementation of the `AsyncActivity` class, whose run member function is invoked from the `async_node`.

```
class AsyncActivity {
  tbb::task_arena a;
public:
  AsyncActivity() {
    a = tbb::task_arena{1,0};
  }
  using async_node_t = tbb::flow::async_node<float, double>;
  using gateway_t = async_node_t::gateway_type;
  void run(float offload_ratio, gateway_t& gateway){
    gateway.reserve_wait();
    a.enqueue([&,offload_ratio]()
      {
        auto t = tbb::tick_count::now();
        // Make triad kernel, NDRange and launch
        auto triad_kernel =
        cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&>
                          {program, "triad"};
        cl::EnqueueArgs q_args{
          queue,
          cl::NDRange{static_cast<size_t>(ceil(vsize*offload_ratio))}
        };
        triad_kernel(q_args, Adevice, Bdevice, Cdevice).wait();
        gateway.try_put((tbb::tick_count::now()-t).seconds());
        gateway.release_wait();
      });
    }
  };
```

Figure 18-11. *AsyncActivity implementation, where the actual GPU kernel invocation takes place*

Instead of spawning a thread as we did in the AsyncActivity of Figure 18-3, we follow here a more elaborated and efficient alternative. Remember that we postponed the explanation of why we used a global_control object in Figure 18-8. In this figure, we initialized the scheduler as follows:

```
tbb::task_scheduler_init init{nth};
auto mp=tbb::global_control::max_allowed_parallelism;
tbb::global_control gc(mp, nth+1); //One more thread, but sleeping
```

If you remember from Chapter 11, the task_scheduler_init line will result in the following:

- A default arena will be created with `nth` slots (one of them reserved for the master thread).
- `nth - 1` worker threads will be populated in the global thread pool, which would occupy the worker slots of the arena as soon as there is work pending in such arena.

But later, the `global_control` object, `gc`, is constructed so that the actual number of workers in the global thread pool is incremented. This extra thread has no slot available in the default arena so it will be put to sleep.

Now, the `AsyncActivity` class, instead of spawning a new thread as we did before, it awakes the dormant thread, which is usually faster, especially if we are invoking several times the `AsyncActivity`. To that end, the constructor of the class initializes a new arena, `a = tbb::task_arena{1,0}`, that has one worker thread slot since it reserves 0 slots for the master. When the member function `run()` is invoked, a new task is enqueued in this arena with `a.enqueue()`. This will result in the dispatch of the dormant thread that will occupy the slot of this new arena and complete the task.

Next, the task spawned in this `AsyncActivity` follows the usual steps to offload computations to a GPU. First, construct the `triad_kernel KernelFunctor` informing that the `triad kernel` has three `cl::Buffer` arguments. Second, call `triad_kernel` passing the `NDRange`, which is calculated as `ceil(vsize*offload_ratio)`, and the GPU view of the buffers, `Adevice`, `Bdevice`, and `Cdevice`.

When running this code on an Intel processor with an integrated GPU, these two lines are generated:

```
Time cpu: 0.132203 sec.
Time gpu: 0.130705 sec.
```

where `vsize` is set to 100 million elements and we have been playing with `offload_ratio` until both devices consume approximately the same time in computing their assigned subregion of the arrays.

Summary

In this chapter, we have first introduced the `async_node` class that enhances the flow graph capabilities when it comes to dealing with asynchronous tasks that escape the flow graph control. In a first simple `Async World` example, we illustrated the use of this class and its companion `gateway` interface, useful to re-inject a message from the asynchronous task back into the flow graph. We then motivated the relevance of this extension to the TBB flow graph, which is easily understood if we realize that blocking a TBB task results in blocking a TBB worker thread. The `async_node` allows for dispatching an asynchronous work outside the flow graph but without blocking a TBB worker thread when waiting for this asynchronous work to complete. We wrapped up the chapter with a more realistic example that puts to work the `async_node` to offload some of the iterations of a `parallel_for` to a GPU. We hope we have provided the basis to elaborate more complex projects in which asynchronous work is involved. However, if we usually target an OpenCL capable GPU, we have good news: in the next chapter, we cover the `opencl_node` feature of TBB, which provides a friendlier interface to put the GPU to work for us!

For More Information

Here are some additional reading materials we recommend related to this chapter:

- Herb Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," www.gotw.ca/publications/concurrency-ddj.htm.
- John McCalpin, STREAM benchmark, www.cs.virginia.edu/stream/ref.html.
- David Kaeli, Perhaad Mistri, Dana Schaa, Dong Ping Zhang. Heterogeneous Computing with OpenCL 2.0. Morgan Kaufmann 2015.