Concurrent CPU-GPU Task Programming using Modern C++

Tsung-Wei Huang* and Yibo Lin†
*Department of Electrical and Computer Engineering, University of Utah
†Department of Computer Science, Peking University

***Abstract*—In this paper, we introduce Heteroflow, a new C++ library to help developers quickly write parallel CPU-GPU programs using task dependency graphs. Heteroflow leverages the power of modern C++ and task-based approaches to enable efficient implementations of heterogeneous decomposition strategies. Our new CPU-GPU programming model allows users to express a problem in a way that adapts to effective separation of concerns and expertise encapsulation. Compared with existing libraries, Heteroflow is more cost-efficient in performance scaling, programming productivity, and solution generality. We have evaluated Heteroflow on two real applications in VLSI design automation and demonstrated the performance scalability across different CPU-GPU numbers and problem sizes. At a particular example of VLSI timing analysis with million-scale tasking, Heteroflow achieved $7.7\times$ runtime speed-up (99 vs 13 minutes) over a baseline on a machine of 40 CPU cores and 4 GPUs.**

I. Introduction

Modern parallel applications in machine learning, data analytics, and scientific computing typically consist of a heterogeneous use of both central processing units (CPUs) and graphics processing units (GPUs) [1]. Writing a parallel CPU-GPU program is never an easy job, since CPUs and GPUs have fundamentally different architectures and programming logic. To address this challenge, the parallel computing community has investigated many programming libraries to assist developers with quick access to massively parallel and heterogeneous computing resources using minimal programming effort [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. In particular, hybrid multi-CPU multi-GPU systems are driving high demand for new heterogeneous programming techniques in support for more efficient CPU-GPU collaborative computing [12]. However, related research remains nascent, especially on the front of leveraging modern C++ to achieve new programming productivity and performance scalability that were previously out of reach [13].

The Heteroflow project addresses a long-standing question: "*how can we make it easier for C++ developers to write efficient CPU-GPU parallel programs?*" For many C++ developers, achieving high performance on a hybrid CPU-GPU system can be tedious. Programmers have to overcome complexities arising out of concurrency controls, kernel offloading, scheduling, and load-balancing before diving into the real implementation of a heterogeneous decomposition algorithm. Heteroflow adopts a new *task-based* programming model using modern C++ to address this challenge. Consider the canonical saxpy (A·X plus Y) example in Figure 1. Each Heteroflow task belongs to one of *host*, *pull*, *push*, and *kernel* tasks; a host task runs a callable object on any CPU core ("the host"), a pull task copies data from the host to a GPU ("the device"), a push task copies data from a GPU to the host, and a kernel task offloads computation to a GPU. Figure 1 explains the saxpy task graph in Heteroflow's graph language.

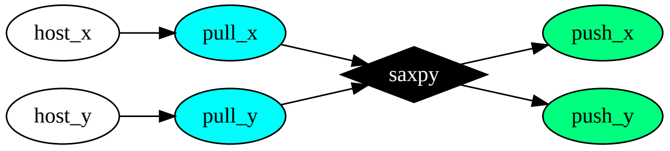

Fig. 1: A saxpy ("single-precision A·X plus Y") task graph using two *host* tasks to create two data vectors, two *pull* tasks to send data to a GPU, a *kernel* task to offload the saxpy computation to the GPU, and two *push* tasks to push data from the GPU to the host.

```
__global__ void saxpy(int n, int a, int *x, int *y){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

const int N = 65536;
vector<int> x, y;

hf::Executor executor;
hf::Heteroflow G;

auto host_x = G.host([&](){ x.resize(N, 1); });
auto host_y = G.host([&](){ y.resize(N, 2); });
auto pull_x = G.pull(x);
auto pull_y = G.pull(y);
auto kernel = G.kernel(saxpy, N, 2, pull_x, pull_y)
                .block_x(256)
                .grid_x((N+255)/256)
auto push_x = G.push(pull_x, x);
auto push_y = G.push(pull_y, y);

host_x.precede(pull_x);
host_y.precede(pull_y);
kernel.precede(push_x, push_y)
      .succeed(pull_x, pull_y);

auto future = executor.run(hf);
```

Listing 1: Heteroflow code of Figure 1.

Listing 1 shows the Heteroflow code that implements the saxpy task graph in Figure 1. The code *explains itself*. The program creates a task dependency graph of two host tasks, two pull tasks, one kernel task, and two push tasks. The kernel task binds to a saxpy kernel written in CUDA [2]. The dependency links form constraints that conform to Figure 1. Heteroflow provides an *executor* interface to perform automatic parallelization of a task graph scalable to manycore CPUs and GPUs. There is no explicit thread managements or fine-grained concurrency controls in the code. Our design principle is to let users write *simple*, *expressive*, and *transparent* parallel code. Heteroflow explores a minimum set of core routines that are sufficient enough for users to implement a broad set of heterogeneous computing algorithms. Our task application programming interface (API) is not only flexible on the user front, but also extensible with the future evolution of C++ standards and heterogeneous architectures. We summarize our contributions as follows:

- **Programming model**. We develop a new parallel CPU-GPU programming model to assist developers with efficient access to heterogeneous computing resources. Our programming model allows users to express a problem with effective separation of concerns and expertise encapsulation. Developers can work at a suitable level of granularity for writing scalable applications that is commensurate with their domain knowledge.
- **Transparency**. Heteroflow is transparent. Developers need not to deal with standard concurrency mechanisms such as threads and fine-grained concurrency controls, that are often tedious and hard to program correctly. Instead, our system runtime abstracts these problems from developers and tackles many of the hardest parallel and heterogeneous computing details, notably resource allocation, CPU-GPU co-scheduling, kernel offloading, etc.
- **Expressiveness**. We leverage modern C++ to design an expressive API that empowers users with explicit graph construction and refinement to fully exploit task parallelism in their applications. The expressive power also lets developers perform rather a lot of work without writing a lot of code. Our user experiences lead us to believe that, although it requires some effort to learn, most C++ programmers can master our APIs and apply Heteroflow to their jobs in just a few hours.

We have applied Heteroflow to two real applications, timing analysis and cell placement, in large-scale circuit design automation and demonstrated the performance scalability across different numbers of CPUs, GPUs, and problem sizes. We believe Heteroflow stands out as a unique tasking library considering the ensemble of software tradeoffs and architecture decisions we have made. With that being said, different programming libraries and frameworks have their pros and cons, and deserve a particular reason to exist. Heteroflow aims for a higher-level alternative in modern C++ domain.

II. Motivation

Heteroflow is motivated by our research projects to develop efficient computer-aided design (CAD) tools for very large scale integration (VLSI) design automation. CAD has been an immensely successful field in assisting designers in implementing VLSI circuits with billions of transistors. It was on the forefront of computing around 1980 and has fostered many prominent problems and algorithms in computer science. Figure 2 demonstrates a conventional VLSI CAD flow with a highlight on physical design. Due to the ever-increasing design complexity, the recent CAD community is driving the need for hybrid CPU-GPU computing to keep tool performance up with the technology scaling [14], [15].

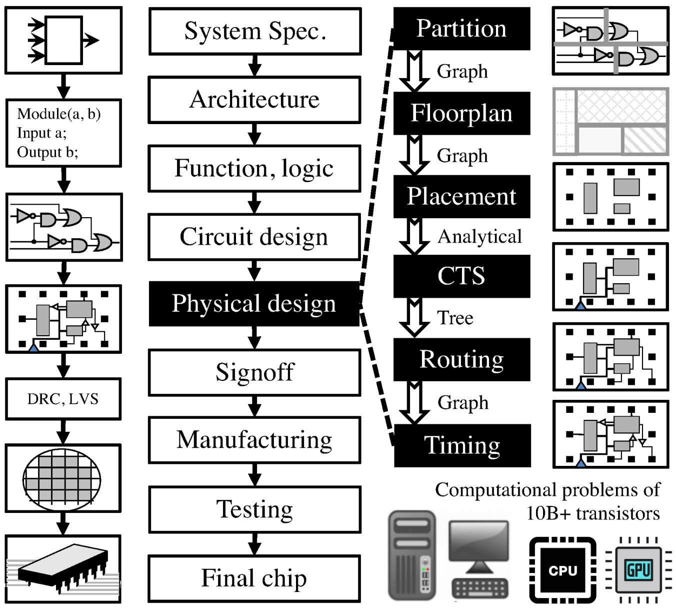

Fig. 2: A typical VLSI design automation flow with a highlight on the physical design stage. Heteroflow is motivated to address the ever-increasing computational need of modern CAD tools.

A. Challenge 1: Vast and Complex Dependencies

Computational problems in CAD are extremely complex and have many challenges that normal software developments do not have. The biggest challenge to develop parallel CAD tools is the vast and complex task dependencies. Before evaluating an algorithm, a number of logical and physical information must arrive first. These quantities are often dependent to each other and are expensive to compute. The resulting task dependency graph in terms of encapsulated function calls can be very large. For example, a million-gate design can produce a graph of billions of tasks and dependencies that takes several days to accomplish [13]. However, such difficulty does not prevent CAD tools from parallelization, but highlights the need of new tasking frameworks to implement efficient parallel decomposition strategies especially with CPU-GPU collaborative computing [15].

B. Challenge 2: Extensive Domain Knowledge

Developing a parallel CAD algorithm requires deep and broad domain knowledge across circuits, modeling, and programming to fully exploit parallelism. The compute pattern is highly irregular and unbalanced, requiring very strategic collaboration between CPU and GPU. Developers often need direct access to native GPU programming libraries such as

CUDA and OpenCL to handcraft the kernels with problem-specific knowledge [2], [3]. Existing frameworks that provide high-level abstraction over kernel programming always come with restricted applicability, preventing CAD engineers from using many new powerful features of the native libraries. Our domain experience concludes that despite nontrivial GPU kernels, *what makes concurrent CPU-GPU programming an enormous challenge is the vast and complex surrounding tasks, most notably the resource controls on multi-GPU cards, CPU-GPU co-scheduling, tasking, and synchronization.*

C. Need for a New CPU-GPU Programming Solution

Unfortunately, most parallel CPU-GPU programming solutions in CAD tools are hard-coded [14], [15]. Developers are "heroic programmers" to handcraft every detail of a heterogeneous decomposition algorithm and explicitly decide which part of the application runs on which CPU and GPU. While the performance is acceptable, it is too expensive to maintain the codebase and scale to new hardware architectures. Some recent solutions adopted directive-driven models such as OpenMP GPU and OpenACC particularly for data-intensive algorithms [6], [7]. However, these approaches cannot handle dynamic workloads since compilers have limited knowledge to annotate runtime task parallelism and dynamic dependencies. In fact, frameworks at functional level are more favorable due to the flexibility in runtime controls and on-demand tasking. Nevertheless, most libraries on this front are disadvantageous from an ease-of-programming standpoint [12]. Users often need to sort out many distinct notations and library details before implementing a heterogeneous algorithm [16]. Also, a lack of support for modern C++ largely inhibits the programming productivity and performance scalability [13], [17]. After many years of research, we and our industry partners conclude the biggest hurdle to program the power of collaborative CPU-GPU computing is a suitable *task programming library*. Whichever model is used, understanding the structure of an application is critical. Developers must explicitly consider possible data or task parallelism of their problems and leverage domain-specific knowledge to design effective decomposition strategies for parallelization. At the same time, the library runtime removes the burden of low-level jobs from developers to improve programming productivity and transparent scalability. To this end, our goal is to address these challenges and develop a general-purpose tasking interface for concurrent CPU-GPU programming.

III. Heteroflow

In this section, we discuss the programming model and runtime of Heteroflow. We will cover important technical details that support the software architecture of Heteroflow.

> *Heteroflow aims to help C++ developers quickly write CPU-GPU parallel programs and implement efficient heterogeneous decomposition strategies using task-based models.*
>
> — Heteroflow's Project Mantra

A. Create a Task Dependency Graph

Heteroflow is *object-oriented*. Users can create multiple task dependency graph objects each representing a unique parallel decomposition in an application. A task dependency graph is a *directed acyclic graph* (DAG) with nodes and edges representing tasks and dependency constraints, respectively. Each task belongs to one of the four categories: *host*, *pull*, *push*, and *kernel*.

1) Host Task: A host task is associated with a *callable* object which can be a function object, binding expression, functor, or a lambda expression. The callable is invoked at runtime by a CPU thread to run on a CPU core. Listing 2 gives an example of creating a host task. In most applications, the callable is described in C++ lambda to construct a *closure* inline in the source code. This property allows host task to enable efficient lazy evaluation and capture any data whether it is declared in a local block or flat in a global scope, largely facilitating the ease of programming.

```
hf::Heteroflow hf;
auto host = hf.host(
  [] () { cout << "task runs on a CPU core"; }
);
```

Listing 2: Creates a host task.

Each time users create a task, the heteroflow object adds a node to its task graph and returns a *task handle* to users. A task handle is a lightweight class object that wraps a pointer to a graph node. The purpose of this extra layer is to provide an extensible mechanism for users to modify the task attributes and, most importantly, prevents users from direct access to the internal graph storage which can easily introduce undefined behaviors. Each node has a general-purpose polymorphic function wrapper to store and invoke different callables according to a task type. A task handle can be empty, often used as a *placeholder* when it is not associated with a graph node. This is particularly useful when a task content cannot be decided until a certain point during the program execution, while the task storage needs preallocation at programming time. These properties are applicable to all task types.

2) Pull Task: A pull task lets users *pull* data from the host to the device. The exact GPU to perform this memory operation is decided by the scheduler at runtime. Developers should think separately which part of their applications runs on which space, and decompose them with explicit task construction. Since most GPU memory operations are expensive compared to CPU counterparts, Heteroflow splits the execution of a GPU workload into three operations, host-do-device (H2D) input transfers, launch of a kernel, and device-to-host (D2H) output transfers, to enable more task overlaps. Pull task adopts this strategy to help users manage the tedious details in H2D data transfers. At the same time, it presents an effective abstraction of which the scheduler can take advantage to perform various optimizations such as automatic GPU mapping, streaming, and memory pooling.

```
vector<int> data1(100);
float* data2 = new float[10];
```

```
auto pull1 = hf.pull(data1);
auto pull2 = hf.pull(data2, 10);
```

Listing 3: Creates two pull tasks.

Listing 3 gives an example of creating two pull tasks to transfer data from the host to the device. The first pull task operates on a C++ vector of integer numbers and the second pull task operates on a raw data block of real numbers. Heteroflow employs the C++20 *span* syntax to implement the pull interface. The arguments forwarded to the pull method must conform to the constructor of `std::span`. In fact, we have investigated many possible data representations and decided to use span because of its lightweight abstraction for describing a contiguous sequence of objects. A span can easily convert to a C-style raw data view that is acceptable by most GPU programming libraries [2], [3], [18]. Sticking with C++ standard also keeps the core of Heteroflow portable and minimizes the rate of change required for our data interface.

```
1  template <typename... ArgsT>
2  auto PullTask::pull(ArgsT&&... args) {
3    get_node_handle().work = [
4      t=StatefulTuple(forward<ArgsT>(args)...)
5    ] (Allocator& a, cudaStream_t s) mutable {
6      auto h_span = make_span_from_tuple(t);
7      auto h_data = h_span.data();
8      auto h_size = h_span.size_bytes();
9      auto d_data = a.allocate(h_size);
10     cudaMemcpyAsync(
11       d_data, h_data, h_size, H2D, s
12     );
13   };
14   return *this;
15 }
```

Listing 4: Implementation details of the pull task.

Listing 4 highlights the core implementation of the pull task based on CUDA. [1] To be concise, we omit details such as error checking and auxiliary functions. The pull task forms a closure that captures the arguments in a custom tuple by which we enable *stateful* task execution (line 4). For instance, in Listing 1, the change made by the host task `host_x` on the data vectors must be visible to the pull task `pull_x`. The stateful tuple wraps references in objects to keep state transition consistent between dependent tasks. Maintaining a stateful transition is a backbone of Heteroflow. Developers can carry out fine-grained concurrency through decomposition and enforce dependency constraints to keep the logical relationship between task data. In terms of arguments, the runtime passes a memory allocator and a CUDA stream to the closure (line 5). The allocator is a pooled resource for reducing GPU memory allocation overhead and the CUDA stream is a sequenced mechanism for interleaving GPU operations [2]. A key motivation behind this design is to support multi-GPU computing. Both the memory allocator and stream are specific to a GPU context which is decided by the scheduler at runtime. Finally, we create a span from the stateful tuple and enqueue the data transfer operation to the stream (line 6:12).

[1]While the current implementation is based on CUDA, our task interface can accept other GPU programming libraries [3].

3) Push Task: A push task lets users *push* data associated with a pull task from the device to the host. The code snippet in Listing 5 creates two push tasks that operate on the pull tasks in Listing 3. The arguments consist of two part, a source pull task of device data and the rest to construct a `std::span` object for the target. Similar to Listing 3, the first push task operates on an integer vector and the second push task operates on a raw data block of floating numbers. Push task is stateful. Any runtime change on the arguments that were used to construct a pull task will reflect on its execution context. This property allows users to create stateful Heteroflow graphs for efficient data management between concurrent CPU and GPU tasks.

```
auto push1 = hf.push(pull1, data1);
auto push2 = hf.push(pull2, data2, 10);
```

Listing 5: Creates two push tasks from the two pull tasks in Listing 3.

```
1  template <typename... ArgsT>
2  auto PushTask::push(PullTask p, ArgsT&&... args){
3    get_node_handle().work = [
4      src=p,
5      t=StatefulTuple(forward<ArgsT>(args)...)
6    ] (cudaStream_t s) mutable {
7      auto h_span = make_span_from_tuple(t);
8      auto h_data = h_span.data();
9      auto h_size = h_span.size_bytes();
10     auto d_data = src.device_data();
11     cudaMemcpyAsync(
12       h_data, d_data, h_size, D2H, s
13     );
14   };
15   return *this;
16 }
```

Listing 6: Implementation details of the push task.

Listing 6 highlights the core implementation of the push task based on CUDA. The push task captures the argument list in the same way as the pull task to form a stateful closure (line 5). The execution context creates a span from the target and extracts the device data from the source pull task (line 7:10). Finally, we enqueue the data transfer operation to a CUDA stream passed by the scheduler at runtime (line 11:13). This CUDA stream is guaranteed to live in the same GPU context as the source pull task. In short, Heteroflow uses pull tasks and push tasks to perform H2D and D2H data transfers. Users explicitly specify the data to transfer between CPU and GPU, and encode these tasks in a graph to exploit task parallelism. They never worry about the underlying details of resource allocation and GPU placement.

4) Kernel Task: A kernel task offloads computation from the host to the device. Heteroflow empowers users with explicit kernel programming using native CUDA toolkits. We never try hard to develop another C++ kernel programming framework that often comes with restricted applicability and performance portability. Instead, users leverage their domain knowledge with the highest degree of freedom to implement their kernel algorithms, while leaving task parallelism to Heteroflow. Listing 7 gives an example of creating two kernel tasks that offload two given CUDA kernel functions to the device using the pull tasks created in Listing 3. The first kernel task operates on `kernel1` with data from `pull1`. The second kernel task operates on `kernel2` with data from `pull2`. Both tasks configure 256 CUDA threads in a block. Kernel functions are not obligated to take any Heteroflow-specific objects. This largely increases the portability and testability of Heteroflow, especially for applications that heavily use third-party kernel functions written by domain experts.

```
__global__ void kernel1(int* data, int N);
__global__ void kernel2(float* data, int N);
auto k1 = hf.kernel(kernel1, pull1, 100)
           .grid_x(N/256)
           .block_x(256);
auto k2 = hf.kernel(kernel2, pull2, 10);
           .grid(N/256, 1, 1)
           .block(256, 1, 1);
```

Listing 7: Creates two kernel tasks that operate on the two pull tasks in Listing 3.

```
1  template <typename F, typename... ArgsT>
2  auto KernelTask::kernel(F&& f, ArgsT&&... args) {
3    gather_sources(args...);
4    get_node_handle().work = [
5      k=*this, f=forward<F>(f),
6      t=StatefulTuple(forward<ArgsT>(args)...)
7    ] (cudaStream_t s) mutable {
8      k.apply_kernel(s, f, t);
9    };
10   return *this;
11 }
12
13 template <typename T>
14 auto KernelTask::gather_sources(T&&... tasks) {
15   if constexpr(is_pull_task<T>) {
16     (get_node_handle().add_sources(tasks), ...);
17   }
18 }
19
20 template<typename F, typename T>
21 auto KernelTask::apply_kernel(
22   cudaStream_t s, F f, T t
23 ) {
24   const auto N = tuple_size<T>::value;
25   apply_kernel(s, f, t, make_index_sequence<N>{});
26 }
27
28 template<typename F, typename T, size_t ... I>
29 auto KernelTask::apply_kernel(
30   cudaStream_t s, F f, T t, index_sequence<I ...>
31 ) {
32   auto& h = get_node_handle();
33   f<<<h.grid, h.block, h.shm, s>>>(
34     convert(get<I>(t))...
35   );
36 }
```

Listing 8: Implementation details of the kernel task.

Listing 8 highlights the core implementation of the kernel task. The kernel method takes a kernel function written in CUDA and the rest arguments to invoke the kernel (line 1:2). The arity must match in both sides. A key difference between Heteroflow and existing models is the way we establish data connection – *we use pull tasks as the gateway rather than raw pointers*. This abstraction largely improves safety and transparency in scaling graph execution to multiple GPUs. From the input argument list, we gather all relevant pull tasks to this kernel (line 3 and line 13:18) and let the scheduler perform automatic device placement. Similar to push and pull tasks, we capture the argument list in a stateful tuple (line 6) and use two auxiliary functions to invoke the kernel from the tuple (line 20:36). All the runtime changes on the arguments will reflect on the execution context of the kernel.

```
1  struct PointerCaster {
2    void* data {nullptr};
3    template <typename T>
4    operator T* () {
5      return (T*)data;
6    }
7  };
8
9  template <typename T>
10 auto KernelTask::convert(T&& arg) {
11   if constexpr(is_pull_task<T>) {
12     return PointerCaster{arg.data()};
13   }
14   else {
15     return forward<T>(arg);
16   }
17 }
```

Listing 9: Implementation details of the data connection between a pull task and a kernel task.

Each argument in the kernel function must experience another conversion (line 34 in Listing 8) before launching the kernel. The purpose of this conversion is to transform the pull task to the type of the corresponding kernel argument, and to possibly conduct any sanity checks at both compile time and runtime. Listing 9 highlights the core implementation of this conversion. The function `convert` evaluates an argument at compile time (line 9:17). If the argument is a pull task, it returns a cast of the internal GPU data pointer to the target argument type (line 11:13). Otherwise, it forwards the argument in return (line 15). The auxiliary structure `PointerCaster` (line 1:7) is designed to operate on plain old data (POD) pointers in support for conventional GPU kernel programming syntaxes. The same concept apply to custom data types depending on a compiler's capability.

5) Add a Dependency Link: After tasks are created, the next step is to add dependency links. A dependency link is a *directed* edge between two tasks to force one task to run before or after another. Heteroflow defines two very intuitive methods, `precede` and `succeed`, to let users create task dependencies. The two methods are symmetrical to each other. A preceding link forces a task to run *before* another and a succeeding link forces a task to run *after* another. Heteroflow's task interface is uniform. Users can insert dependencies between tasks of different types as long as no cycles are formed.

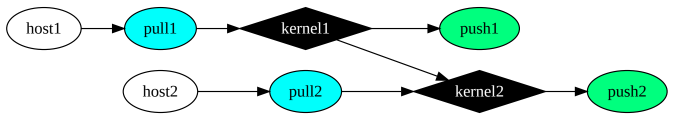

Fig. 3: A task graph of eight tasks and seven dependency constraints.

```
__global__ void k1(int* vec1);
__global__ void k2(int* vec1, int* vec2);

vector<int> vec1, vec2;

hf::Heteroflow hf;
auto host1 = hf.host([](){ vec1.resize(100, 0); });
auto host2 = hf.host([](){ vec2.resize(100, 1); });
auto pull1 = hf.pull(vec1);
auto pull2 = hf.pull(vec2);
auto push1 = hf.push(pull1, vec1);
auto push2 = hf.push(pull2, vec2);
auto kernel1 = hf.kernel(k1, pull1);
auto kernel2 = hf.kernel(k2, pull1, pull2);

host1.precede(pull1);
host2.precede(pull2);
pull1.precede(kernel1);
pull2.precede(kernel2);
kernel1.precede(push1, kernel2);
kernel2.precede(push2);
```

Listing 10: Creates dependency links to describe Figure 3.

Listing 10 gives an example of using the method `precede` to describe the dependency graph in Figure 3. Users can precede an arbitrary number of tasks in one call. The overall code to create dependency links in Heteroflow is very *simple*, *concise*, and *self-explanatory*. An important takeaway here is that task dependency is explicit in Heteroflow. Our API never creates implicit dependency links even though they are obvious in certain graphs. Such concern typically arises when creating a kernel task that requires GPU data from other pull tasks. In this scenario, pull tasks must finish before the kernel task and users are responsible for this dependency in their graphs. Heteroflow delegates the dependency controls to users so they can tailor graphs to their needs. With careful graph construction and refinement, applications can efficiently reuse data without adding redundant task dependencies. For example, `kernel2` in Figure 3 can access the GPU data of `pull1` as a result of transitive dependency (`pull1` precedes `kernel1` and `kernel1` precedes `kernel2`). Listing 10 implements this intent.

6) Inspect a Task Dependency Graph: Another powerful feature of Heteroflow on the user front is the visualization of a task dependency graph using the standard DOT format. Users can find readily available tools such as Python Graphviz and viz.js to draw a graph without extra programming effort. Graph visualization largely facilitates testing and debugging of Heteroflow applications. Listing 11 gives an example of dumping a Heteroflow graph to the standard output.

```
hf.dump(cout);
cout << hf.dump();
```

Listing 11: Dumps a Heteroflow graph to the standard output.

B. *Execute a Task Dependency Graph*

An *executor* is the basic building block for executing a Heteroflow graph. It manages a set of CPU threads and GPU devices to *schedule* in which list of tasks to execute. When a task is ready, the runtime submits the task to an *execution context* which can occur in either a physical CPU core or a GPU device. In Heteroflow, a task is indeed a callable. When users create a task, Heteroflow marshals all required parameters along with unique placeholders for runtime arguments to form a closure that can be run by any CPU thread. Execution of a GPU task will be placed under a GPU context. The scheduler manages all such details to ensure consistent results across multiple GPUs. Listing 12 creates an executor of eight CPU threads and four GPUs and uses it to execute a graph one times, 100 times, and multiple times until a stopping criteria is met. Users can adjust the number based on hardware capability to easily scale their graphs across different CPU-GPU configurations. All the run methods in the executor class are *non-blocking*. Issuing a run on a graph returns immediately with a C++ *future* object. Users can use it to inspect the execution status of the graph or chain up a continuation for asynchronous controls. The executor class also provides a method `wait_for_all` that blocks until all running graphs associated with the caller executor finish. Heteroflow's executor interface is *thread-safe*. Touching an executor from multiple threads is valid. Users can take advantage of this property to explore higher-level parallelism without concerning about race in execution.

```
hf::Executor executor(8, 4); // 8 CPU threads 4 GPUs
hf::Heteroflow graph;
auto future1 = executor.run(graph);
auto future2 = executor.run_n(graph, 100);
auto future3 = executor.run_until(graph, [&] () {
  return custom_stopping_criteria();
});
executor.wait_for_all();
```

Listing 12: Creates an executor to run a Heteroflow graph.

C. *Scheduling Algorithm*

Another major contribution of Heteroflow is the design of a *scheduler* on top of our heterogeneous tasking interface. Scheduler is an integral part of the executor for mapping task graphs onto available CPU cores and GPUs. When an executor is created with N CPU threads and M GPUs, we spawn N CPU threads, namely *workers*, to execute tasks. Unlike existing works [8], [19], we do not dedicate a worker to manage a target GPU, since all tasks are uniformly represented in Heteroflow using polymorphic functional objects (see Listings 4, 6, and 8). This largely facilitates the design of our scheduler in providing efficient resource utilization and flexible runtime optimizations, for instance, GPU memory allocators, asynchronous CUDA streams, and task fusing.

Our scheduler design is motivated by [13]. When a graph is submitted to an executor, a special data structure called *topology* is created to marshal execution parameters and runtime metadata. Each heteroflow object has a list of topologies to track individual execution status. The executor also maintains a topology counter to signal callers on completion. The communication is based on a shared state managed by a pair of C++ *promise* and *future* objects. The first step in scheduling is *device placement*, mapping each GPU task to a particular GPU device. An advantage of our programming model is implicit data dependencies between a kernel and its pull tasks (see line 3 in Listing 8), through which the scheduler can utilize to place them under the right device. Based on this property, we develop a simple and efficient device placement algorithm using *union-find* and *bin packing* as shown in Algorithm 1. The key idea is to group each kernel with its source pull tasks (line 1:7) and then pack each unique group to a GPU bin with an optimized cost (line 8:14). By default, we minimize the load per GPU bins for maximal concurrency but can expose this strategy to a pluggable interface for custom cost metrics.

Algorithm 1: DevicePlacement

```
1  foreach $t \in tasks$ do
2      if $t.type() == KERNEL$ then
3          foreach $p \in t.source\_pull\_tasks()$ do
4              set_union($t$, $p$);
5          end
6      end
7  end
8  foreach $t \in tasks$ do
9      if $x \leftarrow t.type();\ x == KERNEL$ or $x == PULL$
        then
10         if $r \leftarrow set\_find(t);\ is\_set\_root(r)$ then
11             set_bin_packing_with_balanced_load($t$);
12         end
13     end
14 end
```

After device placement, the scheduler enters a *work-stealing* loop where each worker thread iteratively drains out tasks from its local queue and transitions to a *thief* to steal a task from a randomly selected peer called *victim*. The process stops when an executor is destroyed. We employ work-stealing because it has been extensively studied and used in many parallel processing systems for dynamic load-balancing and irregular computations [20], [21]. When a worker thread executes a task, it applies a *visitor* pattern that invokes a separate method for each task type. Running a host task is trivial, but calling a GPU task must be scoped under the right execution context. Heteroflow provides a *resource acquisition is initialization* (RAII)-style mechanism on top of CUDA device API to scope the task execution under its assigned GPU device. Listing 13 gives the implementation details of invoking a pull task from an executor. All GPU tasks are synchronized through CUDA events (line 4 and line 6).

```
1  void Executor::invoke(unsigned me, Pull& h) {
2    auto [d, s, e] = get_device_stream_event(me, h);
3    ScopedDeviceContext ctx(d);
4    cudaEventRecord(e, s);
5    h.work(get_device_allocator(d), s);
6    cudaStreamWaitEvent(s, e, 0);
7  }
```

Listing 13: Implementation details of invoking a pull task.

While detailing the scheduler design is out of the scope of this paper, there are a few notable items. First, each worker keeps a *per-thread* CUDA stream to enable concurrent GPU memory and kernel operations. Second, our executor keeps a *memory pool* for each GPU device to reduce the scheduling overhead of frequent allocations by pull tasks. We implement the famous Buddy allocator algorithm [22]. Third, our work-stealing loop adopts an adaptive strategy to balance working and sleeping threads on top of available task parallelism. The key idea is to ensure one thief exists as long as an active worker is running a task. At the time of this writing, our scheduler design might not be perfect, but it provides a proof of concept for our programming model and fosters future research opportunities for new algorithms.

IV. Experimental Results

We evaluated the performance of Heteroflow on two real VLSI CAD applications, timing analysis and standard cell placement. Each application represents a unique computation pattern. All experiments ran on a Ubuntu Linux 5.0.0-21-generic x86 64-bit machine with 40 Intel Xeon Gold 6138 CPU cores at 2.00 GHz, 4 GeForce RTX 2080 GPUs, and 256 GB RAM. The timing analysis program is compiled by g++8.2 and nvcc CUDA 10.1 with C++14 standards `-std=c++14` and optimization flags `-O2`. The placement program is compiled under the same environment. Both programs are derived from our open-source projects, OpenTimer [23], [24], [25] and DREAMPlace [26], that consist of complex domain-specific algorithms with more than 10K lines of code over years of development.

A. VLSI Timing Analysis

We applied Heteroflow to solve a VLSI timing analysis problem. Timing analysis is a very important component in the overall design flow (see Figure 2). It verifies the expected timing behaviors of a digital circuit to ensure correct functionalities after tape-out. Among various timing analysis problems, one subject is to find the correlation between different *timing views*. Each each view represents a unique combination of a process variation corner (e.g., temperature, voltage) and an analysis mode (e.g., testing, functional). Figure 4 shows the number of required analysis views increases exponentially as the technology node advances [23], [24]. Timing correlation is not only important for reasoning the behavior of a timer but also useful for building regression models to reduce required analysis iterations.

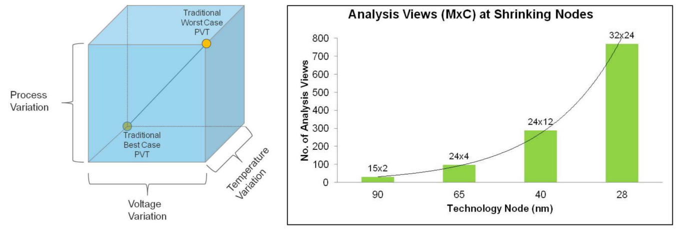

Fig. 4: The required analysis views in terms of corners and modes increase exponentially as the technology node advances.

In reality, there are many ways to conduct timing analysis and correlation. In this experiment, we consider a representative three-step flow: a timer generates analysis datasets from a circuit design across multiple views; a hybrid CPU-GPU algorithm extracts timing statistics and generates regression models for each dataset; a synchronization step combines all assessed quantities to a concrete report. Figure 5 illustrates a fractional task graph of two views. We use the open-source tool, OpenTimer, to generate 1024 different timing reports for a large circuit, *netcard*, of 1.5M gates [23], [24]. The correlation layer implements a CPU-based algorithm to extract graph information (critical paths [27], [28], CPPR [29], [30], [31]) and a GPU-based algorithm to perform logistic regression with gradient descent. Part of CPU and GPU tasks are dependent to each other. For demonstration purpose, we pre-generate the analysis data and control the sample size such that each analysis view takes approximately the same runtime.

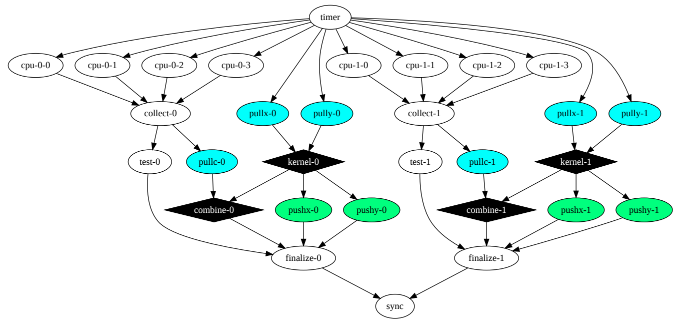

Fig. 5: A partial task graph of VLSI timing analysis for finding correlation between two views. Each view implements a hybrid CPU-GPU correlation algorithm.

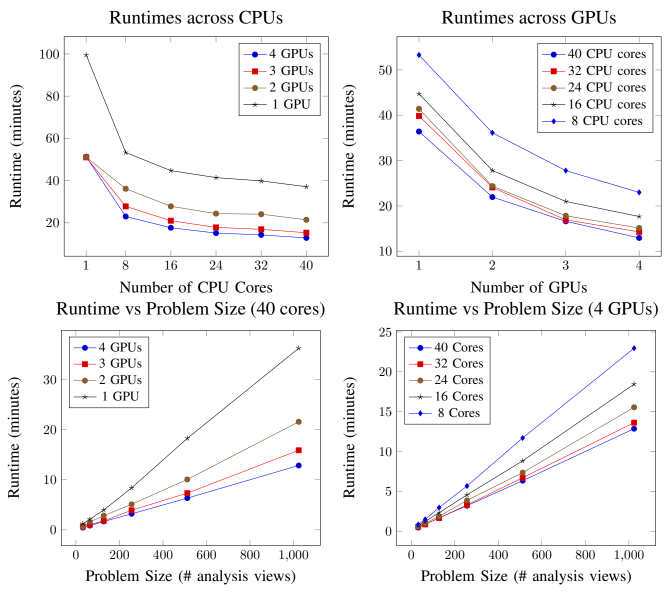

Fig. 6: Runtimes at different CPU-GPU numbers and problem sizes for analyzing the circuit netcard (1.5M gates and 1.5M nets).

Figure 6 shows the overall runtime performance at different CPU-GPU numbers and problem sizes. In general, we observe a descent scaling when increasing the number of cores and GPUs. The task graph requires 99 minutes to finish at the lowest hardware concurrency of 1 core and 1 GPU. Using all 40 cores and 4 GPUs is able to speed up the runtime by 7.7× finishing in 13 minutes. On the slice of 4 GPUs, the runtimes are 51, 23, 18, 15, 14, and 13 minutes for 1, 8, 16, 24, 32, and 40 cores, respectively. The GPU counterparts at 40 cores are 36, 21, 15, and 13 minutes for 1, 2, 3, and 4 GPUs, respectively. The lower side of Figure 6 shows the runtime versus the problem size in terms of six different timing views, 32, 64, 128, 256, 512, and 1024. At any point, increasing the number of CPUs or GPUs can all reduce the runtime. For this particular workload, speed-up from multiple GPUs is more remarkable than CPUs.

B. VLSI Placement

We applied Heteroflow to solve a VLSI placement problem, a fundamental step in the physical design stage (see Figure 2). The goal is to determine the physical locations of cells (logic gates) in a fixed layout region with minimal interconnect wirelength. Modern placement typically incorporates hundreds of millions of cells and takes several hours to finish. To reduce the long runtime, recent work started investigating new algorithms using the power of heterogeneous computing [26]. Among various placement techniques, *detailed placement* is an important step to refine a legalized placement solution for minimal wirelength. Mainstream detailed placement algorithms are combinatorial and iterative. A widely-used matching-based algorithm is shown in Figure 7. The key idea is to extract a maximal independent set (marked in cyan) from a cell set and model the wirelength minimization problem on these non-overlapped cells into a weighted bipartite matching graph. The entire process is very time-consuming especially for large designs with millions of cells. A practical implementation iterates the following three steps: a parallel maximal independent set finding step using Blelloch's Algorithm [32]; a sequential partitioning step to cluster adjacent cells; a parallel bipartite matching step to find the best permutation of cell locations. Figure 7(c) illustrates the process.

In the experiment, we implemented a hybrid CPU-GPU detailed placement algorithm introduced by DREAMPlace [26]. Among these three steps, finding the maximal independent set takes the most runtime. DREAMPlace developed a new acceleration algorithm that offloaded this step to GPU, and showed 40× speed-up over a CPU baseline using 20 cores [26]. The other two steps have graph-oriented computation patterns and are implemented on CPUs. Figure 8 shows a partial task graph for the algorithm in two iterations. The algorithm normally converges in 10-50 iterations. To enable task overlaps between iterations, we flatten the task graph for a given iteration number. The task graph in Figure 8 highlights the complexity of the algorithm and dependent CPU-GPU tasks.

Figure 9 shows the runtime performance at different CPU-GPU numbers and iterations, for placing a large circuit, *bigblue4*, of 2.2M cells and 2.2M nets. It is observed that increasing the number of CPU cores reduces the runtime. For

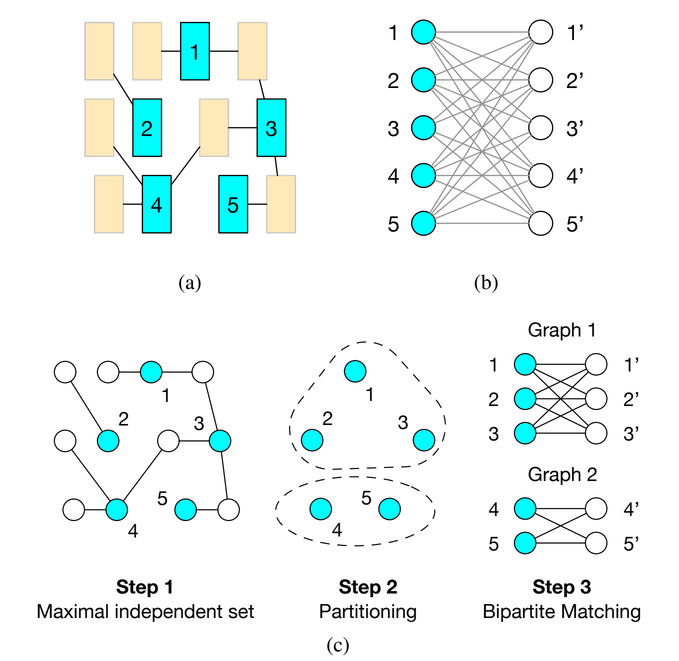

Fig. 7: A matching-based detailed placement algorithm. (a) A placement example of cells and interconnects. Independent cells are marked in cyan. (b) A weighted bipartite matching formulation to find the best permutation of cell locations. (c) A practical three-step iterative implementation of the algorithm.

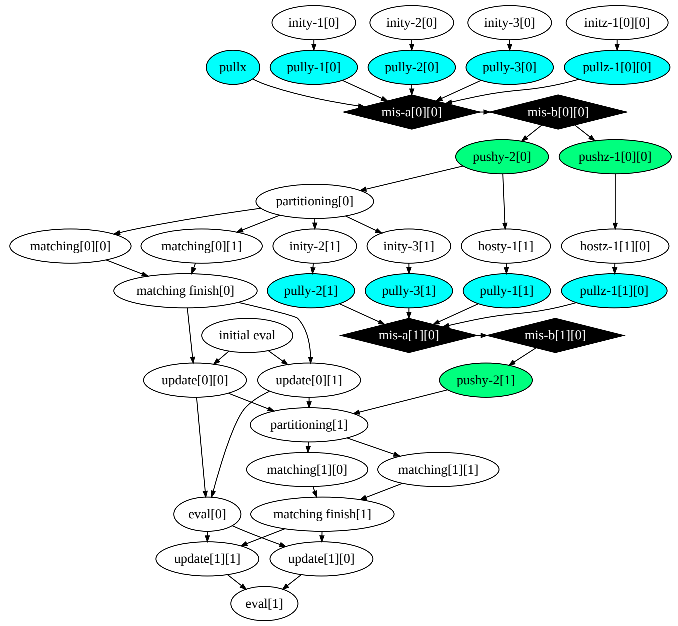

Fig. 8: A partial task graph for the detailed placement algorithm of two iterations. The number in a square bracket indicates the iteration number.

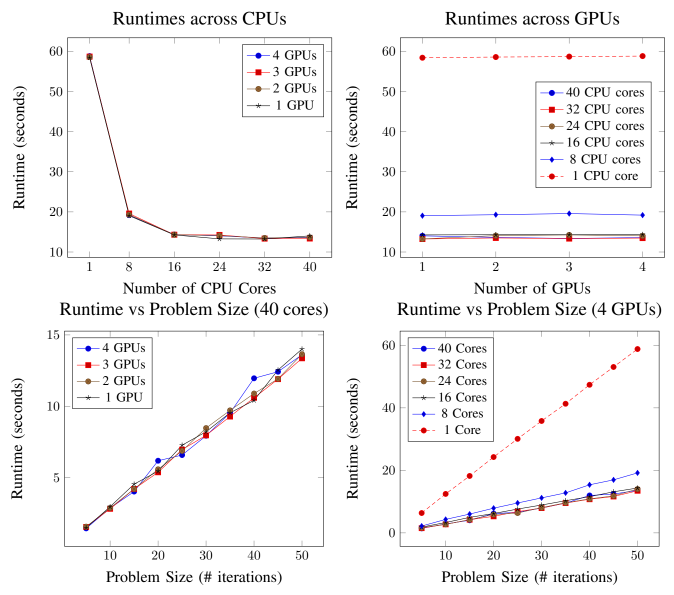

Fig. 9: Runtimes at different CPU-GPU numbers and problem sizes for placing the circuit bigblue4 (2.2M cells and 2.2M nets).

instance, under 1 GPU it takes 58.41s and 14.02s using 1 core and 40 cores, respectively. Maximum concurrency saturates at approximately 20 cores. In this particular workload, performance with 1 GPU is good enough. The runtime does not benefit too much from adding more GPUs. We can clearly see this property on the upper-right plot. Under 40 cores, it takes 14.02s and 13.61s for 1 GPU and 4 GPUs, respectively. In fact, this property is generally true for most optimization algorithms in VLSI CAD, as they are often irregular and dependent [15]. In terms of different problem sizes which is measured by the iteration count used to construct the task graph, increasing the number of CPU cores can reduce the runtime in most scenarios. For example, the task graph of 5 iterations under 4 GPUs finishes in 6.35s and 1.44s using 1 core and 40 cores, respectively. Due to the nature of the algorithm, such trend is not observed on the GPU side.

V. Related Work

Heterogeneous programming models have been extensively developed in scientific communities and enabled vast success in various problem domains [12]. CUDA, OpenCL, OpenGL, C++ AMP, and Brook+ are popular GPU programming frameworks that provide a rich set of low-level APIs for explicit GPU managements [2], [3], [18], [33]. These libraries are designed particularly for power users to implement various optimization strategies specific to a GPU architecture. Directive-based models such as hiCUDA, Ompss, OpenMPC, and OpenACC provide high-level abstraction on GPU programming by augmenting program information, for instance, guidance on loop mapping onto GPU and data sharing rules, to designated compilers [4], [5], [6], [7]. These models are good at loop-based parallelism but cannot handle well irregular task parallelism [34]. Functional-level approaches such as StarPU, SYCL, HPX, PaRSEC, QUARK, XKAAPI++, Unicorn, and Taskflow are capable of concurrent CPU-GPU tasking [8], [9], [10], [11], [19], [35], [16], [17], [36], [37]. The offered graph description languages can be complex or expressive, depending on the targeted applications. Other data structure-driven libraries such as Thrust, VexCL, and Boost.Compute provide C++ STL-style interfaces to program batch CPU-GPU workloads [38], [39], [40]. For concurrent CPU-GPU tasking, users are responsible for scheduling and concurrency controls that are known difficult to program correctly.

CPU-GPU co-scheduling is a pivotal component of all heterogeneous programming systems. The parallel computing community has a number of algorithms including static mapping [41], dynamic work-stealing [20], [21], asymptotic profiling [42], and other system-defined strategies [5], [8], [10], [16]. Vendor-specific features such as CUDA Graph [2], [43] and SYCL [9] offer asynchronous graph scheduling for task parallelism but implementation details are unknown. On the other hand, automatic GPU placement has been studied in machine learning community [44], [45]. The goal is to place operations in a deep neural network onto GPU devices in an optimal way, such that the training process can complete within the shortest amount of time. However, these algorithms are problem-specific and require a unified tensor data structure for performance modeling.

VI. Conclusion

In this paper, we have introduced Heteroflow, a new modern C++ tasking library to help developers quickly write CPU-GPU parallel programs and implement efficient heterogeneous decomposition algorithms. We have evaluated Heteroflow on two real design automation problems and shown performance scalability across different CPU-GPU numbers and problem sizes. At a particular VLSI timing analysis example, Heteroflow can reduce a baseline runtime from 99 minutes to 13 minutes (7.7× speed-up) on a machine of 40 CPU cores and 4 GPUs. Future work will focus on distributing our scheduler based on [46] and incorporating a broader range of workloads, including machine learning [47], [48] and engineering simulation [49], [50], [51].

References

[1] J. S. Vetter *et al.*, "Productive Computational Science in the Era of Extreme Heterogeneity," in *DOE ASCR Report*, 2018.

[2] "CUDA." [Online]. Available: https://developer.nvidia.com/cuda-zone

[3] "OpenCL." [Online]. Available: https://www.khronos.org/opencl/

[4] T. D. Han *et al.*, "hiCUDA: High-level GPGPU Programming," *IEEE TPDS*, vol. 22, pp. 78–90, 2011.

[5] A. Duran *et al.*, "Ompss: a proposal for programming heterogeneous multi-core architectures." *Parallel Processing Letters*, vol. 21, pp. 173–193, 2011.

[6] S. Lee *et al.*, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *IEEE/ACM SC*, 2010.

[7] "OpenACC." [Online]. Available: http://www.openacc-standard.org

[8] E. Agullo *et al.*, "Harnessing clusters of hybrid nodes with a sequential task-based programming model," in *PMAA*, 2014.

[9] "SYCL." [Online]. Available: https://www.khronos.org/sycl/

[10] H. Kaiser *et al.*, "HPX: A task based programming model in a global address space," in *PGAS*, 2014, pp. 6:1–6:11.

[11] G. Bosilca *et al.*, "PaRSEC : A programming paradigm exploiting heterogeneity for enhancing scalability," 2013.

[12] S. Mittal *et al.*, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 69:1–69:35, 2015.

[13] T.-W. Huang *et al.*, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *IEEE IPDPS*, 2019, pp. 974–983.

[14] L. Stok, "Developing Parallel EDA Tools," *IEEE Design Test*, vol. 30, no. 1, pp. 65–66, 2013.

[15] Y.-S. Lu *et al.*, "Can Parallel Programming Revolutionize EDA Tools?" *Advanced Logic Synthesis*, 2018.

[16] T. Beri *et al.*, "The Unicorn Runtime: Efficient Distributed Shared Memory Programming for Hybrid CPU-GPU Clusters," *IEEE TPDS*, vol. 28, no. 5, pp. 1518–1534, 2017.

[17] T.-W. Huang *et al.*, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, vol. 33, no. 6, 2022, pp. 1303 – 1320.

[18] "OpenGL." [Online]. Available: https://opengl.org/

[19] "XKAAPI++." [Online]. Available: http://kaapi.gforge.inria.fr/

[20] J. V. Lima *et al.*, "Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures," *Parallel Computing*, vol. 44, pp. 37–52, 2015.

[21] C.-X. Lin *et al.*, "An Efficient Work-Stealing Scheduler for Task Dependency Graph," in *IEEE ICPADS*, 2020, pp. 64–71.

[22] K. C. Knowlton, "A Fast Storage Allocator," *Commun. ACM*, vol. 8, no. 10, pp. 623–624, 1965.

[23] T.-W. Huang *et al.*, "OpenTimer: A high-performance timing analysis tool," in *IEEE/ACM ICCAD*, 2015, pp. 895–902.

[24] ——, "OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, vol. 40, no. 4, pp. 776–789, 2021.

[25] ——, "OpenTimer v2: A Parallel Incremental Timing Analysis Engine," *IEEE DAT*, vol. 38, no. 2, pp. 62–68, 2021.

[26] Y. Lin *et al.*, "DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement," in *IEEE/ACM DAC*, 2019, pp. 117:1–117:6.

[27] Y. Zamani *et al.*, "A High-Performance Heterogeneous Critical Path Analysis Framework," in *IEEE HPEC*, 2021, pp. 1–7.

[28] K. Zhou *et al.*, "Efficient Critical Paths Search Algorithm using Mergeable Heap," in *IEEE/ACM ASPDAC*, 2022, pp. 190–195.

[29] T.-W. Huang *et al.*, "Fast path-based timing analysis for CPPR," in *IEEE/ACM ICCAD*, 2014, pp. 596–599.

[30] ——, "UI-Timer 1.0: An ultrafast path-based timing analysis algorithm for cppr," *IEEE TCAD*, vol. 35, no. 11, pp. 1862–1875, Nov 2016.

[31] Z. Guo *et al.*, "HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism," in *IEEE/ACM ICCAD*, 2021, pp. 1–9.

[32] G. E. Blelloch *et al.*, "Greedy sequential maximal independent set and matching are parallel on average," in *ACM SPAA*, 2012, pp. 308–317.

[33] I. Buck *et al.*, "Brook for GPUs: Stream Computing on Graphics Hardware," in *ACM SIGGRAPH*, 2004.

[34] S. Lee *et al.*, "Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing," in *IEEE/ACM SC*, 2012.

[35] A. Yarkhan, "Dynamic task execution on shared and distributed memory architectures," *PhD thesis*, 2012.

[36] C.-X. Lin *et al.*, "An Efficient and Composable Parallel Task Programming Library," in *IEEE HPEC*, 2019, pp. 1–7.

[37] ——, "A Modern C++ Parallel Task Programming Library," in *ACM MM*, 2019, p. 2284–2287.

[38] "Thrust." [Online]. Available: https://github.com/thrust/thrust

[39] "VexCL." [Online]. Available: https://zenodo.org/record/571466

[40] "Boost.Compute." [Online]. Available: https://github.com/boostorg/compute

[41] I. Buck *et al.*, "Heterogeneous task scheduling for Accelerated OpenMP," in *IEEE IPDPS*, 2012.

[42] Z. Wang *et al.*, "CPU+GPU scheduling with asymptotic profiling," *Parallel Computing*, vol. 40, no. 2, pp. 107–115, 2014.

[43] D.-L. Lin *et al.*, "Efficient GPU Computation using Task Graph Parallelism," in *Euro-Par*. Springer, 2021, pp. 435–450.

[44] A. Mirhoseini *et al.*, "A Hierarchical Model for Device Placement," in *ICLR*, 2018.

[45] Y. Gao *et al.*, "Post: Device Placement with Cross-Entropy Minimization and Proximal Policy Optimization," in *NIPS*, 2018.

[46] T.-W. Huang *et al.*, "DtCraft: A High-performance Distributed Execution Engine at Scale," *IEEE TCAD*, 2018.

[47] D.-L. Lin *et al.*, "A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism," *IEEE HPEC*, 2020.

[48] D.-L. Lin and T.-W. Huang, "Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism," *IEEE TPDS*, 2022.

[49] Z. Guo *et al.*, "GPU-accelerated Static Timing Analysis," in *IEEE/ACM ICCAD*, 2020, pp. 1–8.

[50] G. Guo *et al.*, "GPU-accelerated Pash-based Timing Analysis," in *IEEE/ACM DAC*, 2021.

[51] ——, "GPU-accelerated Critical Path Generation with Path Constraints," in *IEEE/ACM ICCAD*, 2021, pp. 1–9.