Dear Reviewers,

Thank you so much for your time and efforts in reviewing our submission. Your review comments are very helpful for us to improve the manuscript. Please find our detailed response to your comments in the next pages. We believe this submission has properly addressed your comments.

Response to Reviewer #1

#comment 1: Thank you so much for your comment on the novelty. Pipeline parallelism is not a new parallel model and we do not introduce this parallelism. In this paper, we propose a new framework for users to construct their pipeline applications efficiently. Unlike existing data-centric pipeline frameworks, such as oneTBB and FastFlow, our framework is a task-parallel model. Data-parallel pipeline frameworks consist of data management and scheduling at the same time. We separate these two parts, concentrate on scheduling and let users to manage their data directly. In many CAD workloads, users have their own well-defined data structure and want to apply pipeline parallelism to improve the CAD algorithms. That is, users want to have a full control over their data structure and still have the pipeline parallelism. Pipeflow emerges to fulfill such needs. To our best knowledge, we are the first to separate data management and scheduling. Besides, in Section 4, Pipeflow outperforms the industry standard, oneTBB, in most workloads. Hence, we conclude the novelty of our framework as the followings,

- Pipeflow is the first framework to separate data management and scheduling. Users have full control over their own well-defined data structure and still get the benefit of pipeline parallelism in their applications.

- Pipeflow provides APIs (e.g., tf::line() and tf::pipe()) to query the status of the pipeline. Users could easily know the execution statistics in the middle rather than waiting until the end of the whole execution of the pipeline.

We will revise Section 1 to reflect on your comments and highlight our explanations above.

Response to Reviewer #2

#comment 1: Thank you so much for your comment on the limited form of parallelism. Yes, we agree that pipeline parallelism should combine with other complex parallelism patterns to be effectively used to implement complex parallel applications. We have a VLSI workload of embedding task graph parallelism in each pipe. In addition, we design our pipeline framework as a module, which allows users to exploit pipeline parallelisms with different types of parallelism patterns. We also have examples that combine pipeline parallelism together with other patterns. However, we do not present the VLSI workload and these examples because of the double blind policy. We will revise our appendix to include simple examples that combine both pipeline parallelism and task graph parallelism.

#comment 2: Thank you so much for your comment on the no related work section. We survey some of the famous existing pipeline frameworks in Section 5. The related works contain two parts, pipeline programming models and pipeline scheduling algorithms. For the pipeline programming models, we briefly summarize oneTBB, TPL, GrPPI, FastFlow, TTG, SPar, and Proteas. For the pipeline scheduling algorithms, we mainly talk about oneTBB, GRAMPS, FastFlow, HPX, Cilk-P, FDP, Pipelight, DSWP, LBPP, and URTS. To our best knowledge, these are the most popular pipeline frameworks and we briefly summarize their designs in Section 5. Thanks to the comment of the other reviewer, we will add two more frameworks, OpenMP and CppTaskflow, in Section 5.

#comment 3: Thank you so much for your comment on the no performance comparison with other systems. In Section 4, we study the performance across runtime, memory, and throughput between Pipeflow and oneTBB over micro-benchmarks and two industrial CAD workloads. We use oneTBB as the only one baseline because oneTBB is the industrial standard and has been widely used in the CAD community due to its absolute speed and robustness.

Section 4.2 discusses the micro-benchmarks. The comparisons cover four scenarios: different scheduling token size, different stage size, different core size, and different co-runs. Section 4.3 discusses the VLSI Circuit Timing Analysis workload. The experiments include three scenarios: different stage size, different graph size, and different co-runs. Section 4.4 discusses the VLSI Detailed Placement workload. The experiment covers different core size.

#comment 4: Thank you so much for your comment on the bad scalability performance. In Figure 4, in terms of runtime Pipeflow and oneTBB perform comparably as we increase the scheduling token size. Because oneTBB requires expensive set-up time on the data buffers while starting the pipeline, whereas Pipeflow can immediately start the task scheduling. Yet, as we increase the number of tokens, such cost is amortized. In Figure 5, in terms of runtime the difference decreases gradually. This is because the cost of oneTBB to set up internal data buffers is gradually amortized as we increase the number of stages.

In Figure 9, in terms of memory usage the improvement becomes smaller when the graph size becomes larger. For the VLSI Timing Analysis workload, the pipeline size is proportional to the predefined number of tasks per node and is irrelevant to the graph size. As the graph size increases, it starts to take the majority of memory. In Figure 12, in terms of memory usage the improvement of Pipeflow over oneTBB becomes smaller when the graph size becomes larger. For this VLSI Detailed Placement workload, the pipeline size is proportional to the predefined number of tasks per node and is irrelevant to the graph size. As the graph size increases, it starts to take the majority of memory.

We will revise Appendix and Section 5 to reflect on your comments and highlight our explanations above.

Response to Reviewer #3

#comment 1: Thank you so much for the comment on the English writing. We will check the writing, e.g., missing fill words, using the state-of-the-art tool, Grammarly.

#comment 2: Thank you so much for the comment on the sentence "... flexible framework for users to fully control their application data atop a task-parallel pipeline scheduling framework" in Section 1. The sentence means users want to mange their application data directly and still have the pipeline parallelism in their applications. Current data-parallel pipeline frameworks consist of data management and scheduling at the same time. We separate these two parts, concentrate on scheduling and let users to manage their data directly. In many CAD workloads, users have their own well-defined data structure and want to apply pipeline parallelism to improve the CAD algorithms. That is, users want to have a full control over their data structure and still have the pipeline parallelism. Pipeflow emerges to fulfill such needs.

One metaphor is that oneTBB takes the ownership of user's application data, processes the data in a pipeline fashion, and gives the final results to the user. Before the execution of the pipeline is done, the user can not manage the data because the ownership is in the hand of oneTBB. However, Pipeflow gives a token to a user. By holding the token the user knows what data element should be processed. Since the ownership of the application data is still in the user's hand, he/she could manage the data directly and still have the benefit of pipeline parallelism.

#comment 3: Thank you so much for the comment on the merging of contributions 1 and 2 mentioned in Section 1. The contribution 1 talks about the high-level concept of task-parallel pipeline. The contribution 2 mentions the programming model we introduce to support the concept. The two contributions may have similarities but we like to keep them separated because one is concept and the other one is programming model.

#comment 4: Thank you so much for the comment on the sentence "... users need to sort out a data mapping strategy between their applications and framework-specific abstractions to perform" in Section 1. The sentence means users usually have their well-defined data structure and if oneTBB users want to manage their application data during the execution of pipeline, they need to define another data buffer inside each pipe function and align the data buffer with the global data buffer. Pipeflow provides APIs for users to query the pipeline statistics, which directly shows what data element should be processed. We will revise the sentence to clarify our expression.

#comment 5: Thank you so much for the comment on the sentence “"... update its timing data from a custom, application-dependent circuit graph data structure in a global scope” in Section 2.2. Yes, you are right. Pipeflow allows users to directly manage the intermediate results. oneTBB would give users the final results only. We will revise the sentence to make our expression more clearly.

#comment 6: Thank you so much for the comment on the sentence “From user’s standpoint, the real need is a pipeline scheduling framework to help schedule and run tasks on input tokens across parallel lines …” in Section 2.2. oneTBB is data-parallel pipeline model. Data-parallel pipeline frameworks consist of data management and scheduling at the same time. We separate these two parts, concentrate on scheduling and let users to manage their data directly. In many CAD workloads, users have their own well-defined data structure and want to apply pipeline parallelism to improve the CAD algorithms. That is, users want to have a full control over their data structure and still have the pipeline parallelism. Pipeflow emerges to fulfill such needs.

One metaphor is that oneTBB takes the ownership of user’s application data, processes the data in a pipeline fashion, and gives the final results to the user. Before the execution of the pipeline is done, the user can not manage the data because the ownership is in the hand of oneTBB. However, Pipeflow gives a token to a user. By holding the token the user knows what data element should be processed. Since the ownership of the application data is still in the user’s hand, he/she could manage the data directly and still have the benefit of pipeline parallelism.

#comment 7: Thank you so much for the comment on Listing 2 in Section 3.1. The main differences between oneTBB snippet and Pipeflow snippet are

- oneTBB uses template programming to specify the input type and output type of each pipe. For example, make_filter <float , std :: string> in the second pipe. Pipeflow does not have data abstract and thus does not need template programming to define the data type of each pipe.

- Pipeflow provides APIs for users to query pipeline statistics, such as pf::line() and pf::pipe(). By using these APIs users know which data element to process and can directly manage that data element in user’s application data.

#comment 8: Thank you so much for the comment on Listing 2 in Section 3.1. Yes, the definition of *buf* is used to represent the global data buffer. We will revise the sentence and clarify the purpose to reflect on your comment.

#comment 9: Thank you so much for the comment on the sentence "There are two steps to create a Pipeflow application ... 2) define the data storage" in Section 3.1. The buffer storage is user's application data buffer. In most cases, users have their own well-defined data structure. Hence, the definition of data storage is not needed when users have their global data buffer already. The definition of data storage is application dependent. The most common and straightforward definitions use std::vector. We will revise the sentence to reflect on your comment.

#comment 10: Thank you so much for the comment on the sentence "Users define the number of parallel lines and the abstract function of each stage" in Section 3.1. The interface of definitions of a pipeline structure between oneTBB and Pipeflow is identical. The definition of a pipeline structure of both designs is the following,

```
pipeline(number of lines,
         abstract function of pipe 1,
         abstract function of pipe 2, …)
```

The interface is the same. The implementation of each pipe function is the main difference.

#comment 11: Thank you so much for the comment on the sentence "contains several extensible methods for users to query the runtime statistics" in Section 3.1. The sentence means Pipeflow can provide APIs to meet user's requirement. For example, in a streaming application a user may want to pause the pipeline from execution when streaming data is not ready and resume the execution when streaming data arrives. To meet this requirement, we can add a pause() and a resume() functions in Pipeflow and the user can directly call pf::pause() and pf::resume().

#comment 12: Thank you so much for the comment on the sentence "... we create a one dimensional (1D) array, buf, to store data in uniform storage using std::variant" in Section 3.1. The *buf* is used to represent user's application data buffer. If users have their own pre-defined global data structure, there is no need to define *buf*. We will revise and clarify the purpose of *buf* to reflect your comment.

#comment 13: Thank you so much for the comment on Listing 3 in Section 3.1. Since the pipeline structure may change during runtime, we introduce ScalablePipeline which allows variable assignments of pipes using range iterators. oneTBB can not change the pipeline structure during runtime. Therefore, we do not provide the snippet of oneTBB. This is an advantage of Pipeflow over oneTBB. We will revise and add this advantage in the paper to reflect on your comment.

#comment 14: Thank you so much for the comment on the sentence "... difference between Pipeflow and the baseline is not obvious on small pipelines" in Section 4. Yes, we should define what the baseline is when it is mentioned for the first time. We will revise the sentence to reflect on your comment.

#comment 15: Thank you so much for the comment on the sentence "80 Intel Xeon CPU .." in Section 4. The CPU we use is Intel Xeon Gold 6138. We will revise the sentence to reflect on your comment.

#comment 16: Thank you so much for the comment on the plots in Section 4.3. The speed up bars are used to highlight the runtime improvement of Pipeflow over oneTBB. Without the help of these bars, readers can not easily tell the difference because Pipeflow outperforms oneTBB slightly and the runtime of the two would overlap. We will reproduce consistent plots to reflect on your comment.

#comment 17: Thank you so much for the comment on the sentence "... assisted us in overcoming many programming challenges …" in Section 4.5. Yes, you are right. As CAD researchers and library users, the challenge is the full control of our application data. However, as CAD researchers and library developers, there are several challenges, such as

- data synchronization from one pipe to the next pipe is not easy. When the next pipe starts to fetch a data element stored in the buffer between it and the previous pipe, we need to make sure the next pipe does not fetch the outdated data element.

- data type of each pipe could be different. It is not easy to define buffers for the pipeline when the data type of each pipe is not the same.
- data locality could be difficult to exploit. Since the data types between two pipes could be different, designing a scheduling to take advantage of data locality is a challenge.

There are challenges for library developers. We agree that the sentence "As experienced CAD researchers" should be revised to reflect on your comment.

#comment 18: Thank you so much for the comment on Section 5. Indeed, OpenMp and CppTaskflow are two popular libraries. OpenMP is directive-based model and it exploits loop parallelism with layer-by-layer synchronization. CppTaskflow takes advantage of task graph parallelism. We will either compare our framework with these two models or discuss their designs in the related work.

We will revise Section 1, Section 2, Section 3, Section 4, and Section 5 to reflect on your comments and highlight our explanations above.

Response to Reviewer #4

#comment 1: Thank you so much for the comment on the readability and workflow. Section 2.2 mainly describes the need of task-parallel pipeline parallelism in view of CAD algorithm. Section 3 talks about the details of Pipeflow in terms of programming model and scheduling algorithm. Although Pipeflow is a task-parallel pipeline framework, we should discuss Pipeflow totally in Section 3 and leave Section 2.2 for task-parallel pipeline parallelism only to further increase the readability and workflow of this paper.

#comment 2: Thank you so much for the comment on the related work of adding discussions of pipeline parallelism in different areas. Indeed, we could survey and talk about the applications of task-parallel pipeline parallelism in areas other than CAD. In fact, in Section 2.2 we talk about the need of task-parallel pipeline in PARSEC in the last paragraph. We agree the survey of other areas could be done to support the idea more.

#comment 3: Thank you so much for the comment on the adding of background of micro-benchmarks and CAD algorithms in Section 4. For micro-benchmarks, we simply measure the pure scheduling performance of Pipeflow. The task of each pipe is a combination of simple arithmetic operations and a sleeping thread, which does not cause heavy computations. For VLSI Timing Analysis algorithm, we talk about the algorithm in Section 2.2. Yes, we should add the background of the algorithm in Section 4.3 to increase the readability. For VLSI Detailed Placement algorithm, we describe the algorithm in the first paragraph in Section 4.4 and Figure 11.

#Rebuttal 1: Thank you so much for the question about the difference between current CAD algorithm and our framework. The difference between current CAD algorithm and Pipeflow depends on what programming model the researchers use. Many researchers in the CAD community use directive-based language, such as OpenMP. This loop-parallelization model heavily suffers from synchronization per iteration. Then researchers turn to data flow model, such as oneTBB, and gain huge performance improvement. However, as we describe in Section 1 users want a full control over data. Therefore, we introduce the task-parallel pipeline framework, Pipeflow.

#Rebuttal 2: Thank you so much for the question about the selections of two real-world CAD applications. As experienced parallel CAD researchers, we try to accelerate the execution of CAD applications. VLSI Timing Analysis and VLIS

Detailed Placement are two critical and most time-consuming parts in the back end of VLSI designs. Therefore, we selected these two workloads. In addition to back end, there are front end in VLSI designs. We did not choose one workload from front end because front end usually involves the domain knowledge of hardware description language, which is more suitable in CAD-related conferences (e.g., Design Automation Conference).

We will revise Section 2.2, Section 3, Section 4, and Section 5 to reflect on your comments and highlight our explanations above.