My enthusiasm for teaching started since my high school when teaching lower-grade student algorithms and data structures. Over the past decade, from my dedication to learning and teaching Computer Science as well as mentoring student researchers, I have built my own unique philosophy of education. Instead of conveying knowledge itself, my philosophy is to **inspire** students todevelop their own understandings in an **inclusive**, and **attractive** style so that they all can always learn new things during their rest of lives.

Teaching is all about insipring understandings. During my undergraduate, I had a tough time to learn so many classes. Teachers just read the points on the slides and jump into another one before I understood. I could hardly "memorize" the knowledge without "understanding" the ideas behind them. I had to spend significantly more time after classes to build my own understanding by digging the basic ideas behind. I believe a teacher's mission is to ease the difficulties for students to gain new knowledge by **inspiring** their own understanding. This requires the teachers not only develop deep understandings of the knowledge to convey, but also actively keep track of the students' study progress.

No students shall be left behind. I still clearly remember the first programming quiz in my freshman year. The student backgrounds varied — half of the students already learned programming in high school, and it was the first time for the other half. The first half of students quickly solved all the questions and left early, while the second half of the students were still struggling to ask the TAs about the functions to call, which were not covered neither by the teacher nor the TAs before the quiz at all.

At that moment, I wish I were the teacher so that I would arrange the class in a more **inclusive** style based on the students' diverse backgrounds. This is especially critical for the first two years' undergraduate study. Quizzes, projects, and exams should be developed according to students' prior knowledge level so that no student is left behind.

Keep engaged. Keep updated. Studying sometimes can be really hard, but the pain of study can be eased by making it more **attractive**. According to my personal experience, there are two keys to keep it attractive: creating positive feedback, and keeping the knowledge taught up-to-date. With respect to the positive feedback, I feel really thrilled every time when the program gives results that align with the theory. One example of how I would apply this is that let students program both naive and smart implementations to understand the real world impact of theoretical optimizations. With respect to up-to-date, because of the rapid development of computer science, much knowledge can be deprecated and replaced in less than a decade. During my undergrad teaching, I improved the teaching materials of Compiler Design Implementation by refactoring the course project to a modern programming language.

Teaching during Doctoral Studies During my doctoral studies, I served as a TA for *Computer Systems: A Programmer's Perspective*. In this period, I well practiced my inspiring and inclusive philosophy.

During the TA office hour, when students brought their questions to me, instead of telling them the correct answers directly, I would like to inspire the students to figure out the answers from their own understandings. To specifically figure out what was confusing them, I would work on the questions with the students step by step until we got to the step where they got stuck. Then I would explain the missing knowledge that caused the confusion, and how I understood it when learning. Finally, I would leave the last step, figuring out how these are applied to the questions, to the student.

Moreover, to make sure no student was left behind, I would answer as many questions as possible. If the time ran out, or the questions were too complicated to have instant answers, I would let the student leave me an email so that I could get back to them offline. Also, when approaching each due date of an assignment or lab, I would carefully check the roster to see their status of submission. If some students are missing, I would email them a reminder and ask if they need an extension (with penalties, of course).

Mentoring during Doctoral Studies When mentoring student researchers, I believe the key to inspire their interests of doing research is to ease any unnecessary pains (especially on infrastructure setup) so that they can focus on the innovations.

Therefore, when a new student comes to me, I always first in-person help them setup the research infrastructures. When building the research infrastructures, I would walk through the whole code base from a simple end-to-end example, so that they can understand the role of each code repository in a full-stack system. For the ideas or works I assign to them, I will break my project down to a gentle piece for them to start with by locating the exact modules and lines of the code in the repository. After the work is assigned, I will actively track their progress weekly to see if they need any help or have

progress to report. If so, a meeting can be scheduled to look at the further progress to figure out the next step together.

TA-ship during Undergraduate During my undergraduate studies, I served as a TA for compiler construction. I practiced the philosophy of attraction, by aggressively refactoring the course project to the knowledge up-to-date.

This course used to require students to write a compiler that targets C to MIPS assembly code. C is an old language with many design concepts that are already deprecated in modern languages, and MIPS is not a widely used instruction set either. Meanwhile, during the past decade this course has been taught, the difficulties were increasing unreasonable — all the bonus items achieved last year would be basic requirements next year. I noticed this ill-formed trend and proposed to refactor the course project. My colleagues and I then built a new language manual for the compiler education purpose to better teach the modern compilation principle, and rule out unnecessary difficulties.

For example, because of the limited memory and storage available in the 1980s, the syntax constraints in C guarantee this language can be parsed within one frontend pass, which is no longer true in this century. Therefore, we build a language that should be compiled through multi-pass parsing. This not only attracts students attention by sharing interesting history of computer engineering development, but also teaches students to implement a modern compiler frontend.

Teaching during High School During my high school's last year, I taught students in lower grades (from elementary school to high school) Olympiad Informatics (OI, some Chinese-versioned USACO, or high-school-versioned ACM ICPC). Many students are recommended to top-tier Computer Science programs in China because of their excellent performances in OI contests. I believe that is because my inclusion and attraction philosophy were well practiced.

When preparing my classes for them, I carefully studied all the details of the algorithm or data structure I am going to teach and considered many aspects that students might cast questions. Besides the knowledge itself, I also gathered exercises with gradual difficulties, from straightforward ones to ones with tricks. Students could easily see the performance improvement of a smart algorithm or data structure compared with a brute force implementation on straightforward examples.

When teaching the class, I would set up several checkpoints. When I was done with explaining something, I would stop and interactively ask everyone if everything was clear. This makes sure that everyone is included in the class, and keeps everyone's attention to the teaching.

Classes to Teach To sum up, my research interests were all inspired by the related classes I learned during my undergraduate, including computer architecture, and compiler construction. Thus, to promote this exciting research area, I would like to teach these classes Introduction to Computer Organization, Computer Architecture, and Compiler Construction. Based on my experiences and teaching enthusiasm built when teaching algorithms and data structures in high school, I also would like to teach Discrete Mathematics, and Data Structures and Algorithms. In addition, my future research will be highly related to operating systems, so Introduction to Operating Systems is also under my consideration.

Classes to Develop My research area is highly related to software-hardware co-design innovations. I believe this is a key to the next-generation computing systems, so I would like to open two graduate-level classes related to my research, including one paper-reading seminar on hardware accelerators, and one class on non-conventional programming and execution paradigms.

With regard to the seminar class, I will select a set of papers published in recent top-tier conferences for students to read and present in groups. Each presentation should have three quiz questions to make the presenters think more when reading the paper for good questions and help everyone in this class pay attention to the presentation. The final course project can be proposing a minor improvement to the paper they read, and I will set up a meeting with each group to confirm they are proposing feasible ideas. It is highly encouraged to quantitatively demonstrate the improvements, but it should also be acceptable to have qualitative results.

With regard to the class on non-conventional programming and execution model, in my introduction section, I will first inspire the students to understand the inefficiency of general-purpose units (e.g. CPU), and lead them to realize the rationale and necessity of having specialized hardware. And then, I will let the students do different programming practices, starting with tuning the performance of a CPU and GPU code. I will also apply to both AWS and Google Cloud for the access of their deep neural network accelerators so that the students can try accelerated programming models in data centers and understand the status of hardware specialization in industry.