Episode 17: Expanding the SYCL 2020 Ecosystem with hipSYCL and DPC++
Host: Nicole Huesman, Intel
Guests: Jeff Hammond, Intel; Aksel Alpay, Heidelberg University Computing Center

---

Nicole Huesman: Welcome to *Code Together*, an interview series exploring the possibilities of cross-architecture development with those who live it. I'm your host, Nicole Huesman.

As heterogeneous programming becomes the norm for developers, the adoption of open standards becomes ever more important. And it's truly been amazing to see the growth of the SYCL ecosystem, with, now, four different implementations to support a diversity of hardware architectures: ComputeCpp, triSYCL, hipSYCL and DPC++.

Who better to dive into these than my colleagues, Aksel Alpay and Jeff Hammond?

Aksel is an engineer on the HPC team at Heidelberg University's Computing Center and a member of the Khronos SYCL Working Group. He may be best known as the creator of hipSYCL. Aksel also leads the development efforts of the recently introduced oneAPI Center of Excellence at Heidelberg University. Aksel, so great to have you with us today!

Aksel Alpay: Hi, thanks for having me.

Jeff Hammond is a Principal Engineer at Intel, focused on parallel programming models and HPC workload analysis, as well as HPC business and architecture strategy. He's also deep into SYCL and a big proponent of open standards. Jeff, always great to have you on the program!

Jeff Hammond: Oh, thanks for having me again, Nicole.

Nicole Huesman: So, let's dive in and unpack some of the differences in the SYCL implementations for our listeners, and what hipSYCL and DPC++ bring to the table. All SYCL compilers are based on Clang/LLVM, *but* not in the same way. Aksel, can you help us understand this a little bit more?

Aksel Alpay: Yeah, sure. So, from a 10,000 foot level, the main difference is that if you look at an implementation like DPC++ or ComputeCpp, they have dedicated SYCL front ends that live inside Clang. DPC++ is basically not built on top of Clang, it is Clang, it's a fork of Clang. So, you have this dedicated SYCL front end, which then compiles your SYCL code to whatever you want to compile for, whatever hardware you want to compile for.

In hipSYCL, what we do is something quite different. What we do is we rely on existing compiler tool chains for heterogeneous computing and in particular, the hip tool chain for AMD devices and the CUDA tool chain for Nvidia devices. Both of them are already available in Clang, so what hipSYCL does is that it essentially adds a small-ish Clang plugin on top of these existing tool chains that then teach us the Clang CUDA or hip infrastructure to additionally also accept SYCL code, but it's still the CUDA tool chain and the hip tool chain of Clang.

It truly shows how flexible SYCL is as a model because you can really have these two very different approaches, and both of them work. To understand why we did this approach in hipSYCL, I think it's useful to talk a bit about the history of hipSYCL, why I started it. This has to do with the everyday programming endeavors of a scientist. I have a background in computational astrophysics, and I used to write software for MPI clusters with GPUs and our systems were CUDA-based, but I wanted to be able to support my applications no matter what hardware we are going to have in the future, no matter what hardware I personally am going to have in the future. So, I decided to use open standards. And so, I used OpenCL, but I found myself oftentimes looking enviously to the CUDA side of things because profilers are better for Nvidia devices there and certain hardware features are only exposed in CUDA, but not an OpenCL, so I wondered, could we have an open standard that also allows us to tie into proprietary tool chains if we need to, if you want to do this, this low-level optimization. And then I came across SYCL and I discovered, well, this could actually work. If I build a SYCL implementation on top of something that already exists on top of a proprietary language, such as CUDA, then I end up in something where I can basically, if they have compiling for CUDA and then have an optimized code path for CUDA devices, that then allows me to use all the latest CUDA features, but then also have a regular SYCL code path for all the other devices. And this is exactly what hipSYCL does, so, you can really mix and match CUDA or hip code with your SYCL code, thereby creating highly optimized code paths for specific devices.

Jeff Hammond: So, Aksel, I'm really curious on the timing. Were you one of the first implementations or did you see what was going on for a couple of years?

Aksel Alpay: hipSYCL I started in summer 2018, which means it's certainly not the first implementation, triSYCL and ComputeCpp are much older, but that's when it began. This is also the time roughly when I stumbled across SYCL, when I was sort of looking for a solution for my problem and looking for an open standard that would allow me to easily build something like this.

Jeff Hammond: Now I'm curious, you know, there was OpenMP and there was OpenACC and then there's things like Kokkos. So I'm curious, you know, what are your views on the pros and cons? 'Cause all of them have some pros and some cons. What are your analyses of the different open standards for GPUs?

Aksel Alpay: I think they're complimentary. So OpenMP I think is a great solution if you don't like modern C++ because SYCL is effectively very modern C++. So, there's some people that don't like that, which is fine. OpenMP is also a great solution if you have a large code base that already has lots of sequential codes, lots of loops and so on, and you want to just quickly paralyze that, then you add your pragma and then you're done and you can focus on your science. And so I think OpenMP is a very, very good solution for this, but if you like modern C++, if you create a new code and want it to really integrate well with C++, I think is SYCL as a great choice. Kokkos is also very interesting to us in that respect, and I think between SYCL and Kokkos are very similar. So a major difference here may be, for example, that there are sort of organizational differences, like SYCL is a standard, Kokkos is not a standard. But I think if you program between SYCL and Kokkos, you will have probably similar feeling. One thing that SYCL does very well that I like a lot is this implicit task of building with the accessors. And I think this is also something that's, in this form, almost unique to SYCL in the way that it integrates into the language.

Jeff Hammond: So I'm curious, you know, that's a common pattern, it seems, you know. Have you gone back and written all the astrophysics code in SYCL, or have you found SYCL to be all-consuming enough that astrophysics is on ice for a while?

Aksel Alpay: Well, it's sort of on my to-do list to revisit that code, but I never had the time until now to do that. So, I think this would be a great project in the future, even if it's just for benchmarking or so, to see how it fares with SYCL.

Jeff Hammond: So you mentioned one of the big, important properties of hipSYCL is that it is fully compatible with the native tool chain. I think for me, this feels a lot like the way we use the intrinsics or assembly on traditional languages, you know, x86, for example, people write AVX-512 intrinsics or inline assembly, and, of course, those things are not ISO standard, but people do that. And as you said, they do it with *#ifdef*. So how have you found that to work? Are you writing applications that mix the two? Is it something that is there as a potential, but you find maybe SYCL is actually good enough most of the time?

Aksel Alpay: I think, for many use cases, especially if you're a scientist and you just want to get your program running, you want to focus on the science, I think SYCL is good as it is. It provides the most important primitives for you to work with. But if you're also interested in getting the maximum out of the hardware or using some features that are not yet in SYCL because they're too hardware-specific, or you are a computer scientist and you want to make some experiments with the hardware that you have and you want to really explore it to the fullest, then I think this can be highly interesting. Also it can be highly interesting if you're a library developer. So if your job is to build optimized libraries and you want to have all those optimized code paths for Nvidia devices, AMD devices, Intel devices, whatever.

Jeff Hammond: So, on the library front, I know we've talked in the past a bit about this and this is sort of some of the implementation differences. Can you talk about how hipSYCL integrates with libraries? Are there examples of, with hip or CUDA libraries, that have been done so far? And are there things that, if somebody out there is wanting to build these things, that they should know when they go off to do that?

Aksel Alpay: In principle, it integrates with existing CUDA or hip libraries in the way that every CUDA or hip program would. For example, I've presented some results at IWOCL 2020 with rocPRIM, which is AMD's optimized template library for parallel primitives where if you use rocPRIM in a SYCL kernel with hipSYCL. So, template libraries, so basically this means you compile the AMD hip code into your SYCL program, and it works, and you get some speed up of that because it's AMD-optimized code. There's not much to consider except for how you *#ifdef* exactly, and this is documented in the hipSYCL documentation. If you want to use more higher-level libraries, such as say CuBLAS or so, then you need a way to access the underlying CUDA or hip data types from the hipSYCL runtime, for example, like CUDA stream or hipStream or so, and there are also ways to do that. In particular, in SYCL 2020, there are interoperability mechanisms and they are, for the most part, also supported by hipSYCL.

Jeff Hammond: Cool. So, one of the things I've noticed is that a lot of the projects in the European Exascale program are targeting standards and that's, of course, wonderful, and I know SYCL appears in a lot of contexts. You know, how was your collaboration with either Exascale or other European projects? And how does that align with the various hardware swim lanes that European HPC centers are dealing with? Are you working with any centers, or have you found that they're just using your stuff and it just works?

Aksel Alpay: I'm not directly working with any centers. We have some HPC infrastructure here, and that's also integrated into the larger statewide HPC infrastructure, so I'm working at that level. And apart from that, I collaborate mainly on GitHub and other development channels with the Exascale people.

Jeff Hammond: So you mentioned, you know, how you integrate with Nvidia and AMD native tool chains, and you haven't mentioned, but I of course know that you have an OpenMP tool chain to make the CPU side work. You know, what does it look like to build a new component into hipSYCL, either using, you know, an OpenMP model or a native GPU compiler?

Aksel Alpay: If you wanted to build, say, a new backend there, there are components that you would need to address. So, the first is you need to have support for the runtime side, so you would need to add a new backend for the runtime, which is fairly straightforward. The runtime has C++ base class with virtual functions, these kinds of things, and a model for that, so you just implement that interface and then you're good. If you need some dedicated compiler infrastructure, then you also need to add some compiler driver. We have this SYCL CC compiler wrapper, which can then invoke arbitrary compilers, and to make it nice for the user, it would be a good idea if you also integrate your new compiler there. And lastly, you would potentially need support in the hipSYCL kernel library, so basically to make sure that, for example, the SYCL math functions are mapped to whatever math function your backend would then provide. So, these are the three things that you might want to look at. If you're mainly looking at, for example, a CPU backend, it would be simpler because then the compiler is trivial and the kernel library is also basically just the regular host kernel library, so then you can pretty much focus on the runtime part.

Jeff Hammond: So, we're collaborating in the [Heidelberg University's Computing Center] Center of Excellence. What are you doing for that? And then, I guess, what comes after that in 2021 and beyond for hipSYCL?

Aksel Alpay: So, in the Center of Excellence, we focus on implementing certain key SYCL 2020 features in hipSYCL. These are features that have originated in DPC++ but have now been merged into the SYCL 2020 specification. And this includes features such as unified shared memory (USM), reductions, group algorithms, subgroups, key features that I think most of the SYCL 2020 users are looking forward to. I think these are also features that are highly interesting, in general, because they extend SYCL from a model that was sort of more specific and maybe very good at certain specialized use cases, but was lacking in some degrees to promote it to a true general purpose model, I think, fit for anything that you want to deploy. And I think that's why it's really important that we have widespread implementation coverage of those features.

Beyond that, for SYCL 2021, there are lots of ideas that I have queued up that I want to implement in hipSYCL. hipSYCL has always been an implementation where we have pioneered new or different or maybe slightly weird interpretations of the SYCL standard starting out with, for example, not using OpenCL. Back when hipSYCL was started in 2018, the SYCL specification was hard-coded to rely on OpenCL, and hipSYCL was the first implementation to say, well, in principle it says OpenCL, but it might just as well work with CUDA or hip, so we're just going to do that because there's a benefit. There are some other ideas that I have on the same direction regarding generalization. For example, the meaning of the queue in hipSYCL is already very different, for example, to a queue in most other SYCL implementations because the assumption that a SYCL specification makes is that the queue corresponds to a backend object in some extent, like a CUDA stream or an OpenCL queue and so on.

In hipSYCL, this is not the case. In hipSYCL, the runtime maintains a pool of backend queues—or backend objects might also be other ways of executing tasks—and then the runtime just distributes all incoming work on top of its backend queue pool. And this allows for some optimizations that we can make, regardless of how many SYCL queues the user uses. We can, for example, always guarantee the same overlap of data transfers and compute and these kinds of things, but it also allows for novel work distribution features because the SYCL queue can then be made to be less reliant on the device it was bound to because it's just an interface to the task graph, basically, and the runtime distributes the work anyway. So, there are lots of design things that I want to try out, lots of new features and new interpretations of the SYCL standard.

Jeff Hammond: So, I think the Celerity project, which is independent of hipSYCL but uses it, they have looked at some interesting ideas and distributed stuff. I have some ideas about distributed stuff. You mentioned you started with MPI, of course, which was rather common in astrophysics accounts. What do you see as SYCL beyond just, you know, one GPU, have you thought about that? What does that look like? And how does that interact with the next-generation systems we might see in the world?

Aksel Alpay: I think that SYCL, in general, when it comes to interacting with multiple components, be it multiple backends or multiple devices in the backend, I think it has a highly powerful abstraction with the accessors, because this can allow you to formulate your program in a data flow style in a very easy and intuitive manner. And I think it's not yet ready for true multi-device computation, in particular, because the memory model requires sub buffers for certain guarantees, and that is complicated, but it's also something that hipSYCL doesn't need because hipSYCL memory management also works differently. So, this is also something where I want to spend some time on. In general, I think what the Celerity people do is that they formulate MPI data transfers with an implicit formulation similar to how accessors work with regular SYCL code. And I think this is a highly interesting approach because it allows you to formulate communication in a similar way, regardless of whether it's between multiple devices, multiple nodes, or host and device. And I think this is something that we need to work more with in the future and explore how this concept that we have in SYCL, how this can actually be utilized to its fullest potential.

Jeff Hammond: Now, like me, you're a fan of buffers and accessors and data flow as a very powerful concept for letting, you know, the compiler and runtime do the right thing. Obviously, you're implementing USM, and I think you know, we have a lot of users who really like their pointers. How do you see those two evolving? How do we get the best of both worlds? Are we missing features, or is this just a question of, you know, we have to educate people and help them make the right choices on their designs?

Aksel Alpay: I think USM is an extremely important feature, mainly because of two reasons. One, it allows users to use complex pointer-based data structures in a simple way. And the other is for deployment reasons because there are existing frameworks that assume that there are some pointer-based interface, there are libraries that assume a pointer-based interface, and there are portability tools, for example, the tool to migrate from CUDA to SYCL, which Intel has, would probably be almost impossible to write if there weren't any pointers in SYCL.

Now, what accessors, on the other hand, provide is a very portable way of formulating your memory accessors. If you write a USM program, you either use explicit data copies in which you hard-code, basically, your data copies, which is a problem if, for example, you target CPU, because then those data copies might not even be necessary. Or you use the shared allocations, which migrate automatically, but then the SYCL runtime has no idea what is happening and cannot use information about data transfer sizes or so for its scheduling decisions because it's all handled at a lower level in the driver.

I think there's definitely a huge gap between those two features, and we need to work on how to integrate them. And this could, for example, be that we add additional hints to the USM world to say, okay, runtime, now I'm going to use this part of USM memory and I expect that it will have to be transferred, so please take that into account with your scheduling decision. Or even better, that we sort of find some continuum between the two, sort of like, we construct a buffer with a USM end point, or we have a buffer that operates on USM pointers and we can, if we want, construct accessors from it, but we can also directly operate on the USM pointers. And then do whatever we want, what is more convenient for one kernel, maybe use accessors in one kernel, and use end pointers in the next or so. This is something that I think we need to work much more on in SYCL, and also prototype much more, because I think that no implementation has the full interoperability between the two models yet. And in hipSYCL, we already have buffers that are inherently based on USM pointers, so we can expose much of that, but it's not yet done.

Jeff Hammond: Yeah, that would be a nice feature. I've thought about that at a high level, but that'd be cool to play with in hipSYCL. So, you know, it's funny, I think hipSYCL is often associated with the hip stack, but as you've already told us, you support CUDA and OpenMP. Are you looking at any other devices or any other platforms, or are you focused on the GPU front and then evolving the language?

Aksel Alpay: Yes, I'm always interested in adding more backends and experimenting of how SYCL would work on those backends. For example, I think one thing that's, in particular, very interesting to try out would be an OpenMP 5.0 backend. It's not yet fully clear to what extent this can work in SYCL. There are some things where SYCL exposes more features than OpenMP 5.0, basic things like querying device information. So, this would be something where we would need to experiment to see how far we can take it. But I think that would be highly interesting and also highly useful because it would also allow us to target Intel GPUs, and I think it would be great if hipSYCL could target GPUs from all three vendors.

And looking a bit more in the future, AMD has announced that HIP is going to be also used for their acquired Xilinx FPGAs. So, if this truly turns out to be the case, then I think we could also play with FPGAs in the future with hipSYCL. I think it would be also very, very interesting.

Another thing that I'm highly interested in is backends that are not based on queues, because up to now, most backends that are used in SYCL implementations are based on some sort of queue—CUDA streams, OpenCL queues—but especially if you're on CPU, for example, the task graph backend might be even more interesting, maybe. And there are various options to consider here. Maybe even an existing heterogeneous task graph, a model where you can target both CPUs and GPUs. So, there's lots of room to experiment and improve here.

Jeff Hammond: Is there anything you're looking for from the community? You know, what would people do if they wanted to contribute to the hipSYCL project who is either a user or a Linux maintainer or a vendor or a compiler expert?

Aksel Alpay: Well, the first step, I think, would be to just reach out and get help. If you open an issue there, usually I respond very, very quickly. Then we can work out how we can work together. I'm also always interested in working together with other people. So, if you want to collaborate with me, just let me know and I'm sure we can find some way to give you some nice work in hipSYCL because the project is large, and the team is small. There is a lot to do!

Nicole Huesman: You know, the SYCL ecosystem is such an exciting place right now. And obviously the two of you and many others are really living this journey. Axel, it's great that you're really leading the development at the oneAPI Center of Excellence [at Heidelberg University's Computing Center]. We'd love to have you back on the program to talk more about that.

I'd also like to mention a couple of other things. Axel, you have a panel discussion with Joe Curley, Jeff McVeigh, David Blythe, and so we'll put a link to that panel discussion when we publish this podcast. And then I'd also like to point folks to a presentation and demo that Jeff is doing for GitHub Universe around oneAPI and DPC++ and SYCL. So, we'll also put a link to that as well.

With that, any other pointers as to where folks can go to learn more about SYCL? I know, Axel, you mentioned the different places for hipSYCL. Jeff, for you, how about where folks can go to learn more about DPC++?

Jeff Hammond: There are a couple of different places. My GitHub profile has a couple of different repos. One of the things that I have that might be interesting to folks, especially who are curious about, you know, how does hipSYCL stack up to CUDA or hip, I have this little project with Tim Mattson and a bunch of friends called the Parallel Research Kernels where I recently actually finally added hip support and I ran hip and hipSYCL, and it turns out, hipSYCL performs exactly the same. So, standards don't cost you a thing and you get portability, so you can see that in that code. And I've written a couple of other simple demos. Obviously, Intel has all the oneAPI tutorials. And then, of course, my good friends and colleagues that have written the DPC++ book, and the Apress GitHub has all of the code from that book, and I think you'll link to that, I'm sure. So, there's just so much out there. I'll also plug Tom Deakin. Bristol University has a number of wonderful projects using SYCL and also showing the same thing, which is that hipSYCL performs as well as native APIs, where it supports things. And so, Tom's proved that as well. So, I think it's a great affirmation of what hipSYCL's been doing, and how portability is uncompromising on performance in many ways.

Nicole Huesman: Excellent. Thank you. And, Aksel, any other pointers for listeners in terms of where they can go to learn more?

Aksel Alpay: If you want to learn more about the SYCL ecosystem, a good place to look is also always SYCL.tech where news articles, blog posts, and talks, and these kinds of things are listed about the SYCL ecosystem.

Nicole Huesman: Aksel, thanks so much for being here and for sharing your insights. We look forward to having you back on the program.

Aksel Alpay: Sure. Thank you very much.

Nicole Huesman: And Jeff, it is always such a pleasure to talk to you and to gain your insights, so thanks for being here.

Jeff Hammond: It's always fun. Thank you very much for setting this up.

Nicole Huesman: Absolutely. And for all of you listening, thanks so much for joining us. Let's continue the conversation at oneapi.com. Until next time!