


```

15  argument(pp) { out; element { obj_record; allocate; } } }
16  cuMemcpyDtoHAsync(void* dst, CUdeviceptr src,
17      size_t size, CUstream stream) {
18      async; success(CUDA_SUCCESS);
19      argument(dst) { no_copy; buffer(size); lifetime_manual; }
20      register_buf(async_DtoH, stream, dst, size); }
21  cuLaunchKernel(CUfunction f,
22      uint gridDimX, uint gridDimY, uint gridDimZ,
23      uint blockDimX, uint blockDimY, uint blockDimZ,
24      uint sharedMemBytes, CUstream hStream,
25      void **kernelParams, void **extra) {
26      async; consumes_rc(kernel_invocation, 1);
27      argument(kernelParams) {
28          in; buffer(metadata(f)->fn_argc);
29          element {
30              if (metadata(f)->fn_arg_is_handle[index]) {
31                  type_cast(CUdeviceptr*); buffer(1);
32              } else {
33                  type_cast(void*); buffer(metadata(f)->fn_arg_size[index]);
34              } } }
35      argument(extra) { in; ... } }
36  cuStreamSynchronize(CUstream stream) {
37      contextual_argument void** bufs = get_bufs(async_DtoH, stream);
38      argument(bufs) {
39          in; buffer(get_n_bufs(async_DtoH, stream));
40          element {
41              buffer(get_buf_size(async_DtoH, stream, index));
42              out; deallocate; } } }

```

Figure 4. An example LAPIS description for the CUDA driver API. Code elements in **bold blue** are LAPIS keywords, elements in *italic green* are runtime library calls, elements in gray are function prototypes incorporated by CAVa from the original CUDA header file, and the remaining code is programmer-provided LAPIS. The variable **index** is a LAPIS builtin which provides the index of the current element of a buffer. The specification of the argument *extra* to `cuLaunchKernel` is complex due to alignment and padding, and is elided from this example for clarity. The file `policy_kern.c` is shown in Figure 7.

```

1 CUresult cuStreamSynchronize(CUstream stream) {
2   // Allocate and initialize the command
3   cu_stream_synchronize_call *cmd = allocate;
4   cmd->command_id = CALL_CU_STREAM_SYNCHRONIZE;
5   cmd->call_id = generate_call_id(); // Unique call ID
6   cmd->stream = stream; // Copy simple value
7   // Compute and attach contextual parameter bufs
8   void **bufs = get_bufs(async_DtoH, stream);
9   size_t nbufs = get_n_bufs(async_DtoH, stream);
10  // Attached buffers referenced from bufs
11  void **tmp_bufs = allocate and fill with output buffer sentinel;
12  cmd->bufs = attach_buffer(cmd, tmp_bufs,
13    get_n_bufs(async_DtoH, stream) * sizeof(void *));
14  // Create and register a local record of the call
15  cu_stream_synchronize_record *record = alloc();
16  record->stream = stream; record->bufs = bufs;
17  record->call_complete = 0; register_call(record);
18  // Send the command and wait for and return response
19  send_command(cmd);
20  wait_until(record->call_complete);
21  return record->ret; }
22 CUresult cuMemcpyDtoHAsync(void *dst, CUdeviceptr src,
23    size_t size, CUstream stream) {
24  ...Allocate and initialize the command as above...
25  // Execute state-tracking code from spec
26  register_buf(async_DtoH, stream, dst, size);
27  ...Store and attach arguments similar to above...
28  ...Create and register a local record of the call, and send...
29  // Return the success value from LAPIS specification
30  return CUDA_SUCCESS; }

```

Figure 5. An outline of the generated guestlib code for cuStreamSynchronize and cuMemcpyDtoHAsync. The real code manages a number of additional details, e.g., threads.

上图展示了通过设计的 DSL 写 GPU 加速器代码以及使用 guestlib 生成的代码。

论文中设计的 DSL 叫做 Lapis, Lapis 是一种用于编写 API 规范的高级规范语言。它被设计用于描述计算卸载加速器 API 的行为和语义。以下是 Lapis 的一些特点：

- 高级性：Lapis提供了高级抽象和表达能力，使得开发人员可以描述API的操作、参数、返回值以及它们之间的关系，而无需关注具体的实现细节。这使得规范更易于理解和编写，并且可以促进规范的复用和扩展。
- API特定性：Lapis专门为特定的API设计，例如OpenCL、CUDA和TensorFlow等计算卸载加速器API。它提供了与这些API相关的语法、语义和约束，以便更准确地描述API的行为。
- 自动生成代码：Lapis的规范可以被CAvA编译器自动转换为API特定组件的代码实现。这样，开发人员可以通过编写规范来定义API的行为，而无需手动编写大量的模板代码。这种自动生成代码的方式可以提高开发效率并减少错误。

以上所有的虚拟化方式的粒度要不然是总线级（PCIe）或者 VF 级，一般是整个设备被虚拟化被用户使用，目前没有更加细粒度虚拟化的解决方案。

集群中的虚拟化

在集群上的虚拟化与一个主机差不多，只不过需要通过一个集群管理技术来管理多个集群的资源。kubernetes 提供了服务器集群管理技术，而 kubevirt 则更进一步将虚拟化技术与集群管理技术结合起来可以对服务器集群进行管理，不过异构加速器之间的通信可能会带来一些问题。

不过目前大部分的服务器集群的主流依然是同构加速器，这样更好进行通信以及环境的配置与虚拟机迁移。在这方面也有一些虚拟化相关的研究，但主要是为了在大规模集群中更好利用硬件性能的文章，例如：

- IPADS 的：[Accelerator Virtualization Framework Based on Inter-VM Exitless Communication](#)
- arxiv 上的一篇：[AVEC: Accelerator Virtualization in Cloud-Edge Computing for Deep Learning Libraries](#)

除此之外，一些公司的商业化产品似乎也提到了使用异构加速器来进行 AI 任务：<https://anil-35205.medium.com/ai-virtualization-e384a2454301>