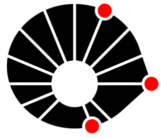

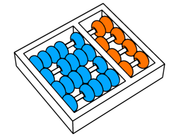

NCCL4JAX: Estendendo JAX com a primitiva AllReduce

S. M. P. Estrela *G. Araújo*

Relatório Técnico - IC-PFG-22-25
Projeto Final de Graduação
2022 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

NCCL4JAX:
Estendendo JAX com a primitiva AllReduce

Sophia Maria Prada Estrela * Guido Araújo†

Julho 2022

Resumo

De um lado temos biblioteca NCCL da NVIDIA em C que traz uma solução rápida para comunicação entre GPUs, entre as funções que ela disponibiliza está a operação AllReduce que é frequentemente usada no treinamento de redes neurais. Do outro temos a biblioteca JAX da Google em Python, que disponibiliza soluções de alto desempenho para machine learning e computação numérica usando o compilador XLA. O objetivo deste projeto é de certa forma unir as duas bibliotecas implementando uma nova primitiva JAX para a operação AllReduce presente na biblioteca NCCL. Após a implementação, uma análise comparativa do tempo de execução da nova primitiva foi feita.

1 Introdução

Com o constante aumento da complexidade dos problemas computacionais modernos, o uso de computação paralela e hardwares dedicados se tornou cada vez mais disseminado. A execução de programas em paralelo se valendo de múltiplas GPUs em clusters, por exemplo, não é mais algo novo para a indústria.

Observando a demanda, a NVIDIA disponibilizou a biblioteca NCCL, uma biblioteca em C que implementa comunicação entre GPUs. Entre as operações implementadas se encontra a operação AllReduce, uma tradicional operação de comunicação coletiva, comumente usada no treinamento de redes neurais.

JAX é uma biblioteca Python de alto desempenho disponibilizada pela Google, seu enfoque é a computação numérica e pesquisa em aprendizado de máquina. Ela trabalha com Autograd e XLA, além de disponibilizar ferramentas para paralelização, vetorização, diferenciação e compilação just in time (JIT)[2].

A intenção deste projeto final de graduação foi implementar uma nova primitiva JAX que englobasse a operação AllReduce da biblioteca NCCL de forma que a operação tivesse um menor tempo de execução e pudesse ser usada aliada a outras ferramentas disponibilizadas pelo JAX.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP
†Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP

2 Conceitos

2.1 JAX

Jax é uma biblioteca Python da Google. Ela permite ao usuário rodar de forma transparente seu código em aceleradores como GPU e TPU e traz a novidade de juntar XLA e Autograd dois projetos anteriores da Google.[2] A biblioteca também traz uma série de ferramentas interessantes para computação de alto desempenho:

- **jax.numpy**: Sua própria versão da biblioteca numpy, a grande diferença é o uso de XLA para compilar e rodar o código em GPUs e TPUs.[2]
- **grad**: Permite que códigos nativos Python ou *jax.numpy* sejam automaticamente diferenciados. A diferenciação é feita da mesma forma que o anterior Autograd.[2].
- **vmap**: "Vectorizing map". Mapeia uma função através de um ou mais eixos. Ou seja, *vmap* pode ser usado para transformar uma função normal em uma função "batched".[2]
- **JIT**: Sigla para compilação "just in time". JAX disponibiliza um decorador e uma função *jit* que permitem que o usuário compile no momento da execução e utilizando XLA juntas as múltiplas operações contidas na função.[2]

2.1.1 XLA

XLA sigla para Accelerated Linear Algebra, é um compilador que foi originalmente disponibilizado para o TensorFlow mas que atualmente é também utilizado em outras bibliotecas como a JAX. Seu diferencial é sua capacidade de gerar otimizações no código de máquina para aceleradores como GPU e TPU. Algumas das otimizações são feitas analisando o grafo de computação criado durante a execução e especializando-o para os tipos e tamanhos dos vetores, outras combinando operações (operation fusion).[3]

2.1.2 JIT

JIT é uma sigla para compilação "just in time", é também uma função e um decorador disponibilizados pela biblioteca JAX. Como diz o nome, JIT dá ao usuário o poder de escolher compilar uma função durante a execução de seu programa.

Ao executar pela primeira vez uma função englobada pelo JIT os seguintes passos acontecem:

1. O JAX recupera todas as operações contidas na função e as envia como um conjunto para o compilador XLA.
2. O XLA compila o conjunto de operações e as otimiza como um todo.
3. O código então compilado é salvo para ser usado nas futuras chamadas a função que acontecerão durante o resto da execução do programa.

Utilizar o JIT porém gera consequências, devido a compilação durante a execução de seu programa a primeira chamada a função englobada pelo JIT tende a ser mais demorada, o benefício aparece porém nas chamadas a função subsequentes, qualquer chamada subsequente tende a ser mais rápida do que o normal, já que o JAX se valerá do código já compilado, otimizado e salvo na primeira execução.

Vale ressaltar também que não é possível usar JIT em todas as funções, por exemplo, não é possível usar JIT em funções que dependem dos valores contidos na entrada para seu fluxo de controle.[2]

2.2 NCCL

NCCL é sigla para "NVIDIA Collective Communications Library", ela é uma biblioteca em C disponibilizada pela NVIDIA. A biblioteca promete, trazer funções para comunicação entre GPUs que são otimizadas pelo uso da biblioteca de somente um Kernel para comunicação e computação e liberar o desenvolvedor da necessidade de otimizar o código para máquina/infraestrutura (GPU) utilizada.[1]

NCCL tenta seguir ao máximo os padrões definidos pela MPI (Message Passage Interface)[1] e traz operações ponto-a-ponto e coletivas comumente usadas nesta, como:

- Allreduce
- Reduce
- AllGather
- Broadcast
- ReduceScatter
- SendReceiv
- One-to-All
- All-to-One
- All-to-All
- Neighbor Exchange

2.2.1 Communicator

Um communicator é uma estrutura utilizada na biblioteca NCCL para auxiliar na comunicação entre GPUs. Cada communicator é associado a um dispositivo além de possuir um ID único.[1]

2.2.2 AllReduce

AllReduce é uma operação coletiva comum no padrão MPI e muito usada no treinamento de redes neurais. Ela também está disponível na biblioteca NCCL para comunicação entre GPUs e é o enfoque do projeto.

A operação AllReduce aplica uma operação de redução nos vetores de entrada de cada rank, e depois faz o broadcast do resultado de volta para cada rank.

As operações de redução disponíveis na NCCL são soma, multiplicação, máximo, mínimo e média.

Se considerarmos que geralmente cada rank fica em um dispositivo diferente, podemos pensar que: A operação AllReduce recupera os valores dos vetores de entrada de **cada** dispositivo, soma os valores contidos nestes (ou aplica a operação de redução passada) e salva o vetor resultado em **cada** dispositivo.

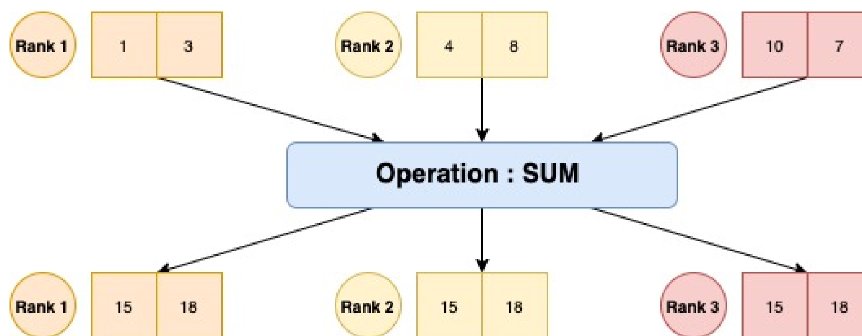

Figura 1: AllReduce com soma como operador (Adaptado de [4])

A figura acima ilustra o que ocorre em um AllReduce com a operação de soma. Cada rank contém um vetor de entrada de mesmo tipo e tamanho, recuperamos os valores contidos nos vetores de cada rank e fazemos a soma dos valores que estiverem na mesma posição:

$$vetor_resultado[i] = vetor_rank_1[i] + vetor_rank_2[i] + vetor_rank_3[i]$$

$$vetor_resultado[0] = 1 + 4 + 10 = 15$$

$$vetor_resultado[1] = 3 + 8 + 7 = 18$$

Com isso, conseguimos gerar um vetor resultado de mesmo tamanho e tipo que os vetores de entrada. Para finalizar, o mesmo vetor resultado que foi encontrado é salvo em cada rank.

3 Implementação

O objetivo do projeto era implementar a biblioteca NCCL4JAX ou seja implementar em forma de primitivas JAX todas operações disponibilizadas na biblioteca NCCL, infelizmente devido ao tempo só foi possível implementar a operação AllReduce.

A fim de criar a primitiva NCCL AllReduce para JAX, foram implementadas três camadas:

- **Camada JAX**: Camada mais perto do usuário, implementada em Python. Nomeada como camada JAX pois será nela que criaremos a primitiva JAX e as funções relacionadas a esta. Também será nela que chamaremos a Custom Call XLA, implementada no que chamaremos camada XLA. Esta camada foi fortemente baseada na biblioteca MPI4JAX.

- **Camada XLA**: Implementada em Cython. É a camada do meio, que une o backend Cython com o código em Python, consiste principalmente da Custom Call XLA. Podemos nos referir também a essa camada como a XLA bridge. Esta camada foi fortemente baseada na biblioteca MPI4JAX.

- **Camada Backend Cython**: O literal backend em Cython. Consiste em uma série de funções retiradas principalmente da biblioteca Cupy assim como arquivos em Cython que fazem a conexão com as bibliotecas NCCL e CUDA disponibilizando algumas de suas funções para utilização em outros arquivos Cython.

Vale a pena comentar, que o código foi fortemente baseado na implementação da biblioteca MPI4JAX, e que grande parte parte da camada Backend Cython foi reaproveitada diretamente da biblioteca Cupy e MPI4JAX.

3.1 Camada JAX

O que chamaremos de camada JAX é a camada mais próxima do usuário, ela é implementada em Python e é onde criamos a primitiva, implementamos a função eval e chamamos a custom call XLA.

O primeiro passo que concluímos é simplesmente criar a primitiva:

```
12    nccl_allreduce_primitive = Primitive("allreduce_nccl")
13    nccl_allreduce_impl = functools.partial(xla.apply_primitive, nccl_allreduce_primitive)
```

Figura 2: Código para criar a primitiva

Depois precisamos fazer o bind dos parâmetros recebidos na função exposta ao usuário à primitiva:

```
15  def allreduce(sendBuf, op,  *, comm=None, token= None):
16
17      if token is None:
18          token = create_token(sendBuf)
19
20      if comm is None:
21          raise Exception("A communicator is needed for this function")
22
23      comm = wrap_as_hashable(comm)
24
25      return tuple(nccl_allreduce_primitive.bind(sendBuf, token, op = op, comm=comm))
```

Figura 3: Função que é exposta ao usuário e que cuida do bind dos parâmetros à primitiva

E por fim precisamos inicializar o ”resto”da primitiva:

```
68  nccl_allreduce_primitive.multiple_results = True
69  nccl_allreduce_primitive.def_impl(nccl_allreduce_impl)
70  nccl_allreduce_primitive.def_abstract_eval(nccl_allreduce_abstract_eval)
71
72  xla.backend_specific_translations["gpu"][nccl_allreduce_primitive] = nccl_allreduce_xla_encode_gpu
```

Figura 4: Código para inicializar a primitiva

Note que na imagem declaramos o seguinte:

1. A primitiva tem múltiplos resultados (L68)
2. A função eval da primitiva é chamada *nccl_allreduce_abstract_eval* (L70)
3. A primitiva terá somente o backend GPU, e este está implementado na função *nccl_allreduce_xla_encode_gpu* (L72).

3.1.1 Múltiplos resultados da primitiva

A nova primitiva/função *allreduce* que implementamos retorna múltiplos resultados (dois) sendo eles o vetor contendo a saída e um token.

A primitiva retorna um token seguindo a lógica apresentada na biblioteca MPI4JAX, aonde eles chegaram a conclusão que o token é necessário quando temos chamadas a funções que dependem do resultado de outra função, por exemplo, *send* e *receive*.[7]

Então, o token retornado pela nova primitiva, poderia ser passado para outra função que dependesse do resultado dela. Vale ressaltar que a primitiva *allreduce* também pode receber um token, tornando possível a lógica oposta.

O token foi criado usando a função *create_token* da biblioteca *jax.lax*.

3.1.2 Função eval da primitiva

Toda primitiva precisa ter uma função eval, nela podemos checar se as entradas seguem os padrões corretos (tipo, tamanho e etc) e retornamos o tamanho e a estrutura da saída da primitiva. A função eval terá como parâmetros os parâmetros da própria primitiva.

A função eval da primitiva *allreduce* é a função nomeada de *nccl_allreduce_abstract_eval*, veja na imagem abaixo como ela foi implementada:

```
61  def nccl_allreduce_abstract_eval(sendBuf, token, op, comm):
62      return (
63          abstract_arrays.ShapedArray(sendBuf.shape, sendBuf.dtype),
64          core.abstract_token,
65      )
```

Figura 5: Função eval

Note que nossa função eval é bem simples, não fazemos nenhuma checagem na entrada, simplesmente retornamos que a saída da primitiva será uma tupla que conterá: um vetor do mesmo tipo e tamanho que o vetor de entrada, e um token.

3.1.3 Implementação da primitiva

A implementação de fato da primitiva é feita na função nccl_allreduce_xla_encode_gpu. Como podemos ver na figura 3 a implementação que falamos é uma implementação para GPU e somente GPU, isso faz sentido se pensarmos que estamos implementando o AllReduce da biblioteca NCCL que faz comunicação **entre GPUs**, não faria sentido termos uma versão desta para CPUs.

Veja na imagem a seguir como a função nccl_allreduce_xla_encode_gpu foi implementada:

```
27  def nccl_allreduce_xla_encode_gpu(stream, sendBuf, token, op, comm):
28
29      comm = unpack_hashable(comm)
30
31      sendBuf_shape = stream.GetShape(sendBuf)
32      sendBuf_dtype = sendBuf_shape.element_type()
33      sendBuf_dims = sendBuf_shape.dimensions()
34
35      nitems = _np.prod(sendBuf_dims, dtype=int)
36      nccl_datatype_code = getNCCLTypeCode(sendBuf_dtype)
37
38      sh = xla_client.Shape.tuple_shape([xla_client.Shape.array_shape(sendBuf_dtype, sendBuf_dims),
39                                         xla_client.Shape.token_shape()])
40
41      descriptor = build_allreduce_descriptor(
42          _np.intc(nitems),
43          nccl_datatype_code,
44          op,
45          comm
46      )
47
48      return xla_client.ops.CustomCall(
49          stream,
50          b"nccl_allreduce",
51          operands=(
52              sendBuf,
53              token,
54          ),
55          shape=sh,
56          opaque=descriptor,
57          has_side_effect=True,
58      )
```

Figura 6: Implementação da primitiva

A parte mais importante da função é a chamada a *xla_cliente.ops.CustomCall* na linha 48, ela chama a custom call registrada na camada XLA. Vamos analisar os argumentos passados a ela:

1. **Linha 49 - stream:** A CUDA stream passada nesta linha para a função *CustomCall* é uma stream criada pelo próprio JAX, porém vale ressaltar que a CustomCall em si ainda é executada na CPU.

2. **Linha 50 - nome da custom call:** O segundo argumento, é o nome dado a custom call na camada XLA.

3. **Linha 51 - operands:** Somente o buffer de entrada e o token são passados como *operands*, pois os operandos devem ficar localizados na GPU.

4. **Linha 55 - shape:** Para o parâmetro *shape* foi passado o formato e tipo da saída.

5. **Linha 56 - opaque:** O parâmetro *opaque* recebe bytes, através dele foram passadas certas informações relevantes para execução mas que não precisam ficar na GPU, como o tamanho e tipo dos buffers, a operação de redução e o communicator.

6. **Linha 57 - has_side_effect:** O último parâmetro simplesmente sinaliza que podemos ter um side effect para a custom call.

3.2 Camada XLA

A parte mais importante da camada XLA é a custom call que podemos ver a seguir:

```
207 cdef void nccl_allreduce_gpu(cudaStream_t stream,  void ** buffers, const char* opaque, size_t opaque_len):
208
209     cdef AllreduceDescriptor* desc = <AllreduceDescriptor*>(opaque)
210     cdef int count = desc.count
211     cdef int datatype = desc.datatype
212     cdef int op = desc.op
213
214     cdef PyObject *ptr_comm
215     ptr_comm = <PyObject *>desc.comm
216     cdef NcclCommunicator comm
217     comm = <NcclCommunicator>ptr_comm
218
219     cdef size_t size
220     size = count * get_type_byte_size(datatype)
221
222     cdef void* in_buf = buffers[0]
223
224     cdef void* out_buf = buffers[2]
225
226     cdef ncclResult_t status
227     status = _ncclAllReduce(<void*>in_buf, <void*>out_buf,
228                                     <size_t>count, <ncclDataType_t>datatype,
229                                     <ncclRedOp_t>op, comm._comm,
230                                     <cudaStream_t>stream)
231
232
233     check_status(status)
234
235     Py_XDECREF(ptr_comm)
```

Figura 7: Implementação da XLA Custom Call

Alguns pontos de interesse na imagem acima:

- Na parte inicial da função (linhas 209 a 217) são recuperados do parâmetro *opaque* os valores que nele foram guardados através do descriptor.

- O buffer de saída (linha 224) é alocado no dispositivo pelo próprio JAX. Ele é o item n+1 (sendo n o número de operandos passados para a custom call) do parâmetro *buffers* (contém os operandos passados para custom call).

- A função *_ncclAllReduce* (chamada na linha 227) chama a função *AllReduce* da biblioteca NCCL, logo depois de recuperar o nome do tipo enviado.

A fim de que a custom call possa ser encontrada, ela precisa ser cadastrada como abaixo no mesmo arquivo que ela foi implementada:

```
241  gpu_custom_call_targets = {}
242
243  cdef register_custom_call_target(fn_name, void* fn):
244      cdef const char* name = "xla._CUSTOM_CALL_TARGET"
245      gpu_custom_call_targets[fn_name] = PyCapsule_New(fn, name, NULL)
246
247
248  register_custom_call_target(b"nccl_allreduce", <void*>(nccl_allreduce_gpu))
```

Figura 8: Registro da XLA Custom Call no arquivo contendo a implementação

E para finalizar o registro da custom call precisamos fazemos o seguinte no arquivo *__init__.py*:

```
4  for name, fn in nccl_xla_bridge.gpu_custom_call_targets.items():
5      xla_client.register_custom_call_target(name, fn, platform="gpu")
```

Figura 9: Registro da XLA Custom Call no arquivo __init__.py

Por fim falaremos do descriptor do allreduce. O descriptor, apesar de ser implementado na camada XLA somente é utilizado na camada Python. Sua função é simplesmente receber os parâmetros que queremos passar via *opaque* para custom call e colocá-los em uma struct que será então transformada em bytes. Ele é necessário pois como discutido na seção 3.1.3, o parâmetro *opaque* da custom call espera receber bytes. Vale ressaltar que a lógica de ter um descriptor para cada primitiva foi retirada da biblioteca MPI4JAX. Veja abaixo como o descriptor do *allreduce* foi implementado:

```
194  cdef struct AllreduceDescriptor:
195      int count
196      int datatype
197      int op
198      PyObject * comm
199
200  cpdef bytes build_allreduce_descriptor(int count, int datatype, int op,   NcclCommunicator comm):
201      Py_XINCREF(<PyObject *>comm)
202      cdef AllreduceDescriptor desc = AllreduceDescriptor(
203          count, datatype, op, <PyObject *>comm
204      )
205      return bytes((<char*> &desc)[:sizeof(AllreduceDescriptor)])
```

Figura 10: AllReduce descriptor

3.3 Camada Backend Cython

A camada backend de Cython consiste em arquivos Cython e cabeçalhos (.h), quase todo o código dessa camada foi retirado da biblioteca Cupy ou MPI4JAX.

Existem quatro arquivos nesta camada que tem os seguintes objetivos:

1. Um arquivo expondo as funções em C da biblioteca CUDA para outros arquivos Cython. Retirado da biblioteca MPI4JAX.

2. Um arquivo com funções que, além de recuperar o tipo correto dado o código passado, simplesmente chamam as funções em C da biblioteca NCCL. Elas servem desta forma para expor as funções C para outros arquivos Cython. Retirado da biblioteca Cupy.

3. Um arquivo declarando um enum que serve para achar o tipo e operação correspondente dado um inteiro. Retirado da biblioteca Cupy.

4. Um arquivo que traz toda infraestrutura NCCL implementada em Cython da biblioteca Cupy. Retirado da biblioteca Cupy.

A infraestrutura NCCL disponível no Cupy foi copiada para que pudéssemos interligá-la com o código criado na camada XLA. Esta foi a única solução encontrada, dentro do tempo disponível para o desenvolvimento do projeto, já que os arquivos Cython que precisávamos não eram expostos pelo Cupy.

Não entraremos em maiores detalhes sobre essa camada, já que ela é quase completamente retirada de outras bibliotecas.

3.4 Exemplo de Uso

Veja um curto exemplo de uso da nova primitiva aliada a JIT:

```
1   import sys
2   sys.path.append('../nccl4jax')
3
4   import jax
5   from jax import jit
6   import jax.numpy as jnp
7   from utils import getNCCLOpCode
8   from xla_bridge.nccl_xla_bridge import NcclCommunicator, _groupStart, _groupEnd
9   from all_reduce_nccl import allreduce
10
11  def run_all_reduce_jitted():
12
13      comms_list = NcclCommunicator.initAll([1,2,3])
14      op =  getNCCLOpCode("NCCL_SUM")
15
16      _groupStart()
17      for comm in comms_list:
18          device_id = comm.device_id()
19          send_buffer = jnp.arange(10, dtype=jnp.int32)
20          jit_all_reduce = jit(allreduce, device=jax.devices()[device_id], static_argnames = ["op", "comm"])
21          recv_buffer, token = jit_all_reduce(send_buffer, op, comm=comm)
22      _groupEnd()
```

Figura 11: Exemplo de uso: nova primitiva allreduce + JIT

Um ponto de interesse neste exemplo é que devido ao fato de termos a infraestrutura NCCL completa do Cupy no nosso backend Cython, quase todas as funções disponíveis em *cupy.cuda.nccl* estão também disponível em *nccl_xla_bridge*. Podemos ver isso pelo uso da função *NcclCommunicator.initAll* originalmente da biblioteca Cupy na linha 13. Vale ressaltar, que isso não significa que poderíamos usar a função diretamente do Cupy, por exemplo a camada XLA espera que o communicator seja criado usando as funções presentes na camada backend Cython.

Outro ponto interessante, é o uso de JIT junto a nova primitiva na linha 20. Note primeiramente que, passamos para função JIT o device, ao fazer isso deixamos para o JAX o trabalho de enviar o *send_buffer* (linha 19) para o device correto, se não utilizarmos o JIT teríamos que fazer este processo manualmente usando por exemplo a função *jax.device_put*.

É importante notar também, a declaração dos parâmetros *op* e *comm* como *static_argnames*, isso é obrigatório para fazer o JIT funcionar com a nova primitiva, e serve para sinalizar que toda vez que um destes dois parâmetros forem alterados precisamos recompilar a função.

4 Resultados

Haviam dois objetivos em implementar a função AllReduce da biblioteca NCCL como uma diretiva JAX:

1. Alcançar um speedup comparado a alternativa existente na biblioteca Cupy
2. Permitir ao usuário da biblioteca JAX, usar a nova primitiva aliada de outras ferramentas disponíveis pela biblioteca como por exemplo JIT.

A fim de comprovar se os objetivos foram alcançados, foram criados três programas:

1. Um usando a nova primitiva sozinha.
2. Um usando a nova primitiva e JIT.
3. Um usando somente a biblioteca Cupy.

Todos os três programas computam a AllReduce usando a operação de soma e através de 3 GPUs diferentes.

O resultado a seguir foi encontrado após remover a primeira execução e tirar a média de 50 execuções:

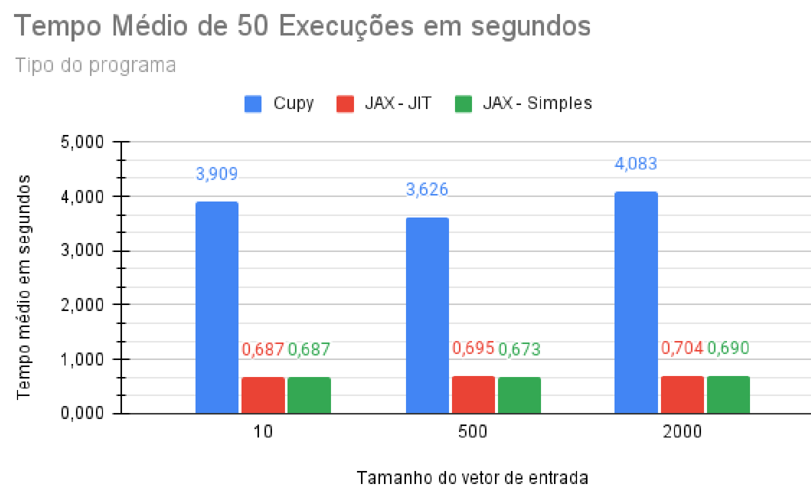

Figura 12: Gráfico contendo a média de 50 execuções em segundos, após retirar a primeira execução

Podemos ver na tabela, que alcançamos com sucesso um speedup de mais de 5x:

Programa	**Tempo Médio**	**Speedup**
Cupy	4,083 s	1,000
JAX - JIT	0,704 s	5,800
JAX - Simples	0,690 s	5,917

Tabela 1: Tabela contendo o tempo médio em segundos e speedup de 50 execuções com um vetor de tamanho 2000

Existe um speedup significativo ao usarmos a primitiva JAX, isso pode ser provavelmente atribuído ao uso da custom call XLA, pois como dito na seção 2.1.1, o compilador XLA é capaz de gerar um código de máquina bastante otimizado.

Outro detalhe interessante, é que o speedup é maior por uma pequena margem no programa que utiliza a primitiva JAX sem o JIT. Isso provavelmente ocorre pois no programa o JIT engloba somente a função/primitiva *allreduce*, o que limita o número de otimizações que XLA pode fazer. Basicamente, se englobarmos somente a função *allreduce* no JIT, não conseguiremos mais otimizações além do que já é feito no JAX - Simples, porém a adição de outras operações na seção dentro do JIT pode possivelmente gerar um speedup maior.

4.1 Primeiras execuções

Foi comentado anteriormente, que para chegar aos resultados apresentados na imagem 12, retiramos o tempo da primeira execução de todos os programas dos cálculos de média. Isso foi feito pois as primeiras execuções dos programas JAX demoram muito mais que qualquer outra execução destes.

O tempo maior na primeira execução dos programas JAX, pode ser provavelmente atribuído ao JAX usando a primeira execução para compilar usando XLA a função e armazenando o resultado desta compilação para usar nas execuções subsequentes. Este comportamento, apesar de esperado para o programa JAX - JIT, é ligeiramente inesperado para o programa JAX - Simple, pois demonstra que a compilação em tempo de execução está ocorrendo mesmo sem o uso do JIT. Veja na tabela abaixo, os tempos encontrados na primeira execução dos programas:

Programa	**Tamanho 10**	**Tamanho 500**	**Tamanho 2000**
Cupy	13,861 s	6,915 s	10,743 s
JAX - JIT	4,442 s	4,784 s	4,242 s
JAX - Simples	4,879 s	4,565 s	4,933 s

Tabela 2: Tempos das primeiras execuções dos programas JAX em segundos

Vale a pena ressaltar que, apesar do tempo da primeira execução dos programas JAX ser muito maior que suas execuções subsequentes, este é bastante similar ao tempo médio de execução do programa Cupy.

5 Conclusão

O objetivo principal do projeto que era implementar uma nova primitiva JAX para a operação AllReduce da biblioteca NCCL foi alcançado. Um programa simples, que executa a um AllReduce com soma em 3 GPUs usando a nova primitiva apresentou um speedup médio de 3,977, quando comparado a um programa que usa a biblioteca Cupy nas mesmas condições.

É possível também usar a nova primitiva aliada junto a função JIT da biblioteca JAX. A execução neste caso, nas mesmas condições, levou a um um speedup médio de 3,855. O speedup menor pode ser atribuído ao fato de só termos a nova primitiva englobada pelo JIT, um aumento no speedup pode ser possivelmente alcançado ao englobarmos mais operações no JIT.

Os speedups previamente mencionados só foram calculados após a remoção do primeiro tempo de execução de todos os programas. Isso foi feito principalmente devido aos programas JAX, pois o uso do compilador XLA leva a primeira execução destes a ter um tempo consideravelmente maior que as execuções subsequentes. Essa discrepância pode ser provavelmente atribuída ao JAX compilando a função e armazenando o resultado para usar nas execuções subsequentes.

5.1 Trabalhos Futuros

Alguns dos possíveis próximos passos do projeto seriam:

- Implementar as outras operações disponíveis na biblioteca NCCL.
- Tornar possível operações in-place. Esse comportamento é permitido na biblioteca NCCL e na biblioteca MPI4JAX, tornando bastante provável que possa ser implementado também na NCCL4JAX.

Referências

[1] NVIDIA Corporation, 2020, *NCCL Documentation website*, accessed 2022-07-12, `https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html`.

[2] Google LLC, 2020, *JAX Documentation website*, accessed 2022-07-12, `https://jax.readthedocs.io/en/latest/`.

[3] XLA and TensorFlow team, 2017, Google LLC, *XLA - TensorFlow, compiled*, accessed 2022-07-12, `https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html`.

[4] Wes Kendall, 2022, MPI Tutorial, *MPI Reduce and Allreduce*, accessed 2022-07-12, `https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/`.

[5] Dion Häfner and Filippo Vicentini, 2021, *MPI4JAX Documentation*, accessed 2022-07-12, `https://mpi4jax.readthedocs.io/en/latest/`.

[6] Google, 2021, *TensorFlow Custom Call XLA Documentation*, accessed 2022-07-14, https://www.tensorflow.org/xla/custom_call.

[7] Häfner, D. & Vicentini, F. mpi4jax: Zero-copy MPI communication of JAX arrays. *Journal Of Open Source Software.* **6**, 3419 (2021)

[8] dfm, 2021, *Extending JAX with custom C++ and CUDA code*, accessed 2022-07-18, https://dfm.io/posts/extending-jax/