# RWKV6 Vector算子设计文档

Sunyuqing 30040711

202410

1. **设计目标**
   1. 场景与基本说明

实现rwkv6的time mix单元recurrent模式下的计算逻辑：

for b in range(B):

    for h in range(H):

        for i in range(N):

            state = np.zeros((N), dtype=data\_type)

            for t in range(T):

                for j in range(N):

                    x = k[b, t, h, j] \* v[b, t, h, i]

                    s = state[j]

                    o[b, t, h, i]+=r[b, t, h, j] \* (u[h, j] \* x + s)

                    state[j] = s \* w[b, t, h, j] + x

接口：

extern "C" \_\_global\_\_ \_\_aicore\_\_ void rwkv6\_vector(uint32\_t B, uint32\_t T, uint32\_t C, uint32\_t H, GM\_ADDR k, GM\_ADDR v, GM\_ADDR w, GM\_ADDR r, GM\_ADDR u, GM\_ADDR o, uint32\_t tileLength)

**详细入参说明：**

* B：输入的Batch数
* shape：[]
* 类型为 uint32
* T：输入的序列长度
* shape：[]
* 类型为 uint32
* C: 输入的维度
* shape：[]
* 类型为 uint32
* H: 输入的attention数，H能被C整除, N = C // H
* shape: []
* 类型为 uint32
* k: 输入矩阵k
* shape：[B, H, T, N]
* Datatype: half;
* v: 输入矩阵v
* shape：[B, H, T, N]
* Datatype: half
* w: 输入矩阵w
* shape：[B, H, T, N]
* Datatype: half
* r：输入矩阵r
* shape：[B, H, T, N]
* Datatype: half
* 数据格式：ND
* u: 输入矩阵u
* shape：[H, N]
* Datatype: half
* o: 输入、输出矩阵o
* shape：[B, H, T, N]
* Datatype: half
* tileLength: 输入的T维度tiling参数
* shape：[]
* Datatype: uint32\_t
  1. 数据类型

目前的实现版本中只涉及half类型，后续根据需要将wkv计算部分转换为fp32精度

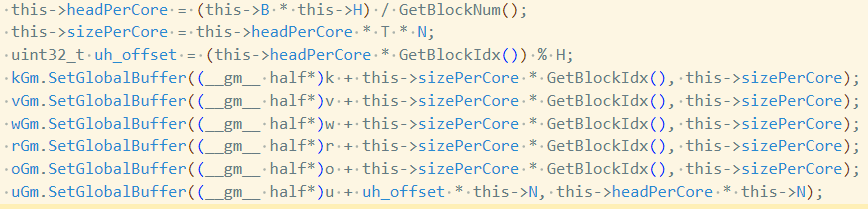
1. **算子实现方案**

**2.1分核，Tiling，数据搬运**

输入矩阵为k, v, w, r, u, o, 其中k, v, w, r, o矩阵的shape为 [B, H, T, N], u矩阵的shape为[H, N];输出矩阵为o;

因此选择在B和H维度进行分核，对k，v, w, r, o在T维度进行tiling切分后分tile搬运。

**2.1.1分核**：当前代码版本中实现B \* H可以被核数整除的情况，每个核上处理 B \* H/coreNum 份数据。矩阵u只有[H, N]两个维度，需要计算当前的核对应的h。



**2.1.2 Tiling：**对数据排布为[B, H, T, N]顺序的k, v, w, r, o矩阵，在T维度切分后，每次搬运的数据块大小为 tileLength \* N 个元素。 具体tileLength可以根据N的大小、UB内存192KB、DataCopy指令单次搬运量越大连续datablock越长性能越好，等综合考虑。

**2.1.3 其他Tiling方案**

矩阵k, v, w, r, o如果按照[B, T, H, N]的数据排布顺序，则另一种方案是在对H维度进行Tiling切分，每次搬运的数据块大小为 h\_tileLength \* N个元素。但此种情况下，由于计算过程需要同一个state矩阵(N \* N) 从t=1迭代到t=T，则在UB上需要存放 k \* h\_tileLength \* N \* N份内存作为计算过程的中间变量 （k为中间变量的个数）。如果h\_tileLength过小，则会导致搬运指令较过多影响性能。

因此综合考虑后选择将H维度与T维度对换，并且选择在T维度进行切分，此种情况下在UB上需要存放的中间变量为k \* N \* N，tileLength可以选择较大的值, 提高搬运指令性能和mte2 mte3带宽利用率。

**2.2 算法流程**：

算子输入：k, v, w, r, o [tileLength, N]由GM搬运至UB；

输出：o[tileLength, N] 由UB搬运至GM；

tileLength为T维度tiling参数；

state中间变量：[N, N] 存放于UB上

对下述步骤重复tileLength遍：

1. 首先获得当前t对应的kt, vt, wt, rt, 获得当前h对应的uh
2. 对kt进行broadcast操作：[1, N] -> [N, N], 对vt进行broadcast操作：[N, 1] -> [N, N], 进行vector乘操作，kv = kt \* vt 存放于UB上
3. 首先获得当前t对应的kt, vt, wt, rt, 获得当前h对应的uh
4. 对uh进行broadcast操作，[1,N] -> [N, N], 并进行vector乘操作，ukv = uh \* kv 并存放于UB上
5. 计算 sukv = state + ukv ，结果存放于UB上
6. 对wt进行broadcast操作：[1, N] -> [N, N], 并更新state = state \* wt 结果存放于UB上
7. 更新state = state + kv ，结果存放于UB上
8. 对rt进行broadcast操作: [1, N] -> [N, N], 并计算 out = rt \* sukv , 结果存放于UB上
9. 对out进行ReduceSum操作，ot = reducesum(out, dim=1) , 将ot赋值给o矩阵对应位置

**2.3 相关代码**

**2.3.1 Pytorch伪代码**

for b in range(B):

    for h in range(H):

        for i in range(N):

            state = np.zeros((N), dtype=data\_type)

            for t in range(T):

                for j in range(N):

                    x = k[b, t, h, j] \* v[b, t, h, i]

                    s = state[j]

                    o[b, t, h, i]+=r[b, t, h, j] \* (u[h, j] \* x + s)

                    state[j] = s \* w[b, t, h, j] + x

**2.3.2 Pytorch算子实现等价代码**

以下代码为当前算子实现的计算逻辑在pytorch上的等价代码。

输入上将k,v,w,r,o都做了H和T维度的rearrange：

k = rearrange(k, 'b t h n -> b h t n')

使得输入的k, v, w, r, o矩阵都按照 B，H，T，N的维度顺序连续排列

**def** rwkv6\_vector\_torch(B, T, C, H, k, v, w, r, u, o):

    """

    input: k,v,w,r,u,o

    output: o

    k, v, w, r, o: shape [B, H, T, N]

    u: shape [H, N]

    """

    N = C // H

    for b in range(B):

        for h in range(H):

            uh = u[h].unsqueeze(0).broadcast\_to((N, N))

            state = torch.zeros((N, N), dtype=torch.float16)

            for t in range(T):

                kt = k[b, h, t].unsqueeze(0).broadcast\_to((N, N))

                vt = v[b, h, t].unsqueeze(1).broadcast\_to((N, N))

                rt = r[b, h, t].unsqueeze(0).broadcast\_to((N, N))

                wt = w[b, h, t].unsqueeze(0).broadcast\_to((N, N))

                kv = torch.mul(kt, vt)

                ws = torch.mul(wt, state)

                state = torch.add(ws, kv)

                ukv = torch.mul(uh, kv)

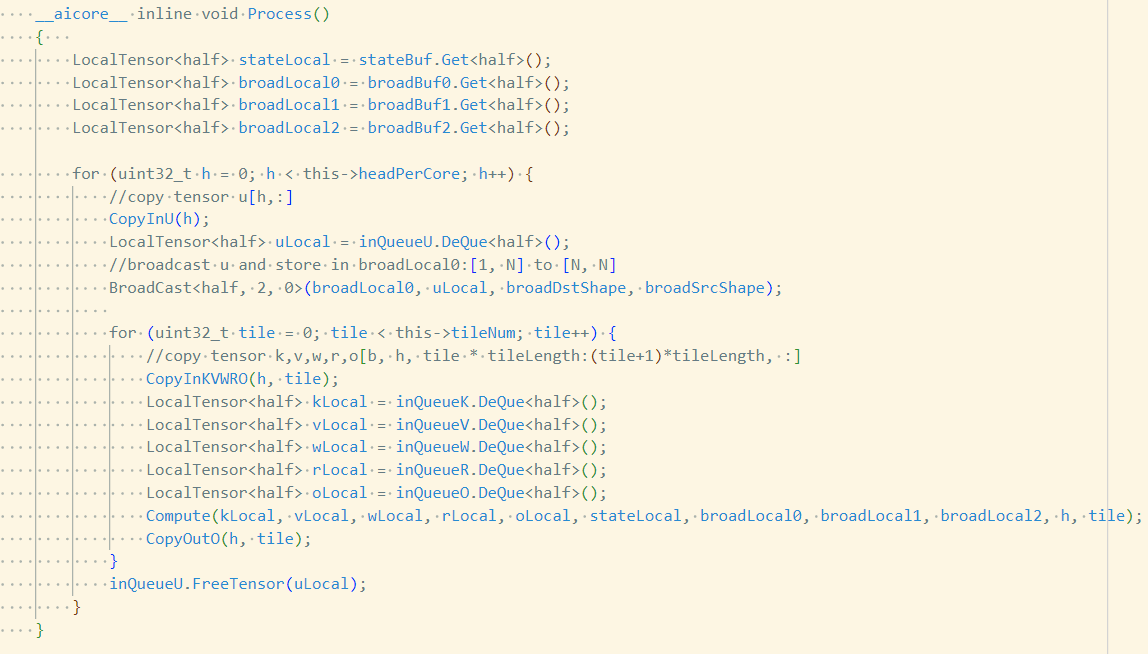
                out\_r = torch.mul(rt, ukv)

                out = torch.sum(out\_r, dim=1)

                o[b, h, t] = out

    return o

**2.3.3 算子实现：Process函数**



**2.3.4 算子实现：Compute函数**

    \_\_aicore\_\_ inline void Compute(LocalTensor<half> kLocal, LocalTensor<half> vLocal, LocalTensor<half> wLocal, LocalTensor<half> rLocal, LocalTensor<half> oLocal, LocalTensor<half> stateLocal, LocalTensor<half> broadLocal0, LocalTensor<half> broadLocal1, LocalTensor<half> broadLocal2, uint32\_t progress\_h, uint32\_t progress\_tile)

    {

        uint32\_t offset0 = 0; //reserved for state vectors

        uint32\_t offset1 = this->N \* this->N;

        uint32\_t offset2 = this->N \* this->N \* 2;

        if (progress\_tile == 0) {

            Muls(stateLocal[offset0], stateLocal[offset0], (half)0, this->N \* this->N);

        }

        for (uint32\_t t=0; t < this->tileLength; t++) {

            //compute kv = k.mT@v, offset1

            //broadcast v from [N,1] to [N, N]

            BroadCast<half, 2, 1>(broadLocal2, vLocal[t\*this->N], vDstShape, vSrcShape);

            //broadcast k from [1,N] to [N, N]

            BroadCast<half, 2, 0>(broadLocal1, kLocal[t\*this->N], broadDstShape, broadSrcShape);

            Mul(stateLocal[offset1], broadLocal1, broadLocal2, this->N \* this->N);

            PipeBarrier<PIPE\_V>();

            //compute ukv = u \* kv, shape: N \* N, offset2, u was stored in broadLocal0

            Mul(stateLocal[offset2], broadLocal0, stateLocal[offset1], this->N \* this->N);

            PipeBarrier<PIPE\_V>();

            //compute sukv = state + ukv, shape:N \* N, offset2

            Add(stateLocal[offset2], stateLocal[offset2], stateLocal[offset0], this->N \* this->N);

            PipeBarrier<PIPE\_V>();

            //compute state = w \* state, shape:N \* N, state

            //broadcast w from [1, N] to [N, N]

            BroadCast<half, 2, 0>(broadLocal1, wLocal[t\*this->N], broadDstShape, broadSrcShape);

            Mul(stateLocal[offset0], broadLocal1, stateLocal[offset0], this->N \* this->N);

            PipeBarrier<PIPE\_V>();

            //compute state = state + kv, shape:N\*N, state

            Add(stateLocal[offset0], stateLocal[offset0], stateLocal[offset1], this->N \* this->N);

            //compute out = r \* sukv, shape:N \* N, offset2

            //broadcast r from [1, N] to [N, N]

            BroadCast<half, 2, 0>(broadLocal1, rLocal[t\*this->N], broadDstShape, broadSrcShape);

            Mul(stateLocal[offset2], broadLocal1, stateLocal[offset2], this->N \* this->N);

            PipeBarrier<PIPE\_V>();

            //compute reduceSum(out), shape: N

            //mask=N, repeatTimes=N, dstRepStride=1, srcBlkStride=1, srcRepStride=N\*sizeof(half)/32=4

            WholeReduceSum(oLocal[t\*this->N], stateLocal[offset2], this->N, this->N, 1, 1, this->N\*sizeof(half)/32);

        }

        //move o from vecin to vecout then free vecin o

        LocalTensor<half> oOutLocal = outQueueO.AllocTensor<half>();

        DataCopy(oOutLocal, oLocal, this->tileLength \* this->N);

        outQueueO.EnQue<half>(oOutLocal);

        inQueueO.FreeTensor(oLocal);

        //free k,v,w,r vecin for reuse

        inQueueK.FreeTensor(kLocal);

        inQueueV.FreeTensor(vLocal);

        inQueueW.FreeTensor(wLocal);

        inQueueR.FreeTensor(rLocal);

    }

**2.4 AIV UB Buffer内存**

输入输出：

k,v,w,r,o 各搬入大小为tileLength \* N 的部分；o同时也是输出矩阵因此还需要一个大小为tileLength \* N的buffer存放输出结果

u矩阵搬入大小为N 的部分

中间变量：

计算过程中需要存放 3份大小为 N \* N 的矩阵；

另外需要3个大小为N \* N的buffer用于存放各个broadcast的结果。