CuPy_MultiLimb_

June 10, 2025

[99]:
```
!pip install cupy
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: cupy in ./.local/lib/python3.11/site-packages
(13.4.1)
Requirement already satisfied: numpy<2.3,>=1.22 in
/opt/conda/lib/python3.11/site-packages (from cupy) (1.26.4)
Requirement already satisfied: fastrlock>=0.5 in ./.local/lib/python3.11/site-
packages (from cupy) (0.8.3)
```

[100]:
```
!pip install cryptography
import cupy as cp
import numpy as np
import time

##Dependencies for the code written
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: cryptography in /opt/conda/lib/python3.11/site-
packages (42.0.5)
Requirement already satisfied: cffi>=1.12 in /opt/conda/lib/python3.11/site-
packages (from cryptography) (1.16.0)
Requirement already satisfied: pycparser in /opt/conda/lib/python3.11/site-
packages (from cffi>=1.12->cryptography) (2.22)
```

[105]:
```
#Parameters to use in the following
LIMB_BITS = 64
LIMB_MASK = (1 << LIMB_BITS) - 1
NUM_LIMBS = 2 # 256-bit numbers with 64-bit limbs

def pad_limbs(arr, target_limbs):
    diff = target_limbs - arr.shape[1]
    if diff > 0:
        padding = cp.zeros((arr.shape[0], diff), dtype=cp.uint64)
        return cp.concatenate([arr, padding], axis=1)
    return arr

def limb_mul(a, b):
```

```
    """
    a-cp.uint64
    b-cp.uint64
    """
    batch = a.shape[0]
    k_a = a.shape[1]
    k_b = b.shape[1]
    result = cp.zeros((batch, k_a + k_b), dtype=cp.uint64)
    for i in range(k_a):
        for j in range(k_b):
            prod = a[:, i] * b[:, j]
            result[:, i + j] += prod & LIMB_MASK
            if i + j + 1 < k_a + k_b:
                result[:, i + j + 1] += prod >> LIMB_BITS
    return result
def limbs_to_int(limbs):
    val = 0
    for i in reversed(range(limbs.shape[0])):
        val = (val << LIMB_BITS) | int(limbs[i])
    return val

def int_to_limbs(x, num_limbs):
    return cp.array([(x >> (64 * i)) & 0xFFFFFFFFFFFFFFFF for i in␣
 ↪range(num_limbs)], dtype=cp.uint64)

def batch_limbs_to_int(batch_limbs):
    ints = []
    for limbs in batch_limbs:
        ints.append(limbs_to_int(limbs.get()))
    return ints
def batch_int_to_limbs(int_list, num_limbs):
    """ a list of Python ints to a 2D CuPy array
    """
    LIMB_MASK = (1 << 64) - 1
    batch = []
    for x in int_list:
        limbs = [(x >> (64 * i)) & LIMB_MASK for i in range(num_limbs)]
        batch.append(limbs)
    return cp.array(batch, dtype=cp.uint64)

def barrett_reduce_gpu(x, n, mu):

    batch_size = x.shape[0]
    k = n.shape[1]
   ## print(f"x.shape={x.shape}, n.shape={n.shape}, mu.shape={mu.shape}")
    #q1 = x[:, (k-1):]
    if x.shape[1] < 2 * k:
```

```
        x = pad_limbs(x, 2 * k)
    q1 = x[:, k-1:-1]
    mu = pad_limbs(mu, x.shape[1])
    q1 = pad_limbs(q1, x.shape[1])
    # q2 = q1 * mu
    q2 = limb_mul(q1, mu)
   ### print(f"q2: {[hex(int(i)) for i in q2[0].get()]}")
    # Taking upper limbs only
    q3 = q2[:, -(k+1):]
    #print(f"q3: {[hex(int(i)) for i in q3[0].get()]}")
    # q3 * n
    q3n = limb_mul(q3, n)
    # r = x - q3 * n
    x_int = limbs_to_int(x[0].get())
    q3n_int = limbs_to_int(q3n[0].get())
    r_int = x_int - q3n_int
    if r_int < 0:
        r_int += (1 << (64 * x.shape[1]))
    r = int_to_limbs(r_int, n.shape[1])
    if r.ndim == 1:
        r = r[cp.newaxis, :]
###    print(f"r : {[hex(int(i)) for i in r[0].get()]}")
    result_int = r.dot(1 << cp.arange(n.shape[1] * LIMB_BITS, step=LIMB_BITS))
    n_int = int("".join([f"{x:016x}" for x in reversed(n[0].tolist())]), 16)
    r_int = r_int % n_int
    r = int_to_limbs(r_int, n.shape[1])
    if r.ndim == 1:
        r = r[cp.newaxis, :]
 ##   print(f"Final result: {[hex(int(i)) for i in r[0].get()]}") added for␣
 ↪debug
    return r

##MODULAR MULTIPLICATION IMPLEMENTATION
def mod_mul(a, b, n, mu):
    x = limb_mul(a, b)
    r = barrett_reduce(x, n, mu)
    return r

##BARRETT IMPLEMENTATIION
def modexp_barrett_gpu(base, exp, n, mu):
    # base, n, mu: (1, num_limbs) cp.uint64 arrays
    # result: (1, num_limbs) cp.uint64
    num_limbs = base.shape[1]
    result = cp.zeros((1, num_limbs), dtype=cp.uint64)
    result[0, 0] = 1  # Set to 1 in limb form

    base_copy = cp.copy(base)
```

```
    while exp > 0:
        if exp & 1:
            prod = limb_mul(result, base_copy)
            result = barrett_reduce_gpu(prod, n, mu)
        prod = limb_mul(base_copy, base_copy)
        base_copy = barrett_reduce_gpu(prod, n, mu)
        exp >>= 1
    return result
def compute_barrett_mu(n_int, k):
    """Compute Barrett mu = floor(b^{2k} / n)"""
    b = 1 << 64
    mu = (1 << (2 * k * 64)) // n_int
    return mu
```

[106]:
```
def mod_inverse_cupy(a, m):
    a = int(a)
    m = int(m)
    m0 = m
    t, new_t = 0, 1
    r, new_r = m, a

    while new_r != 0:
        quotient = r // new_r
        t, new_t = new_t, t - quotient * new_t
        r, new_r = new_r, r - quotient * new_r
    if t < 0:
        t += m0

    return cp.int64(t)
```

[113]:
```
## To benchmark the perfomance, we decided to take
##differnet limb sizes, with other params same
##and check how much time encryption and ecryption takes

p = 61
q = 53
n = p*q
e = 17
#d = 2753
d = mod_inverse(e,(p-1)*(q-1))
print(d)

for num_limbs, msg in zip([1, 2, 3], [42, 124, 1234]):
    mu_int = compute_barrett_mu(n, num_limbs)
    msg_cp = batch_int_to_limbs([msg], num_limbs)
    n_cp = batch_int_to_limbs([n], num_limbs)
    mu_cp = batch_int_to_limbs([mu_int], num_limbs * 2)
```

```
    # Encryption timing
    start_enc = time.time()
    cipher_cp = modexp_barrett_gpu(msg_cp, e, n_cp, mu_cp)
    cp.cuda.Device(0).synchronize()
    end_enc = time.time()
    # Decryption timing
    start_dec = time.time()
    plain_cp = modexp_barrett_gpu(cipher_cp, d, n_cp, mu_cp)
    dev = cp.cuda.Device()
    cp.cuda.Device(0).synchronize()
    id_dev = cp.cuda.Device().id
    props = cp.cuda.runtime.getDeviceProperties(id_dev)
    print(f" GPU usedd {id_dev}: {props['name'].decode()}")

    end_dec = time.time()

    dec_ints = batch_limbs_to_int(plain_cp)
    correct = (dec_ints[0] == msg)

    print(f"Limbs: {num_limbs} | Message: {msg} | Match: {correct}")
    print(f"Encryption : {end_enc - start_enc:.6f} seconds")
    print(f"Decryption : {end_dec - start_dec:.6f} seconds")
```

```
2753
 GPU usedd 0: NVIDIA GeForce GTX 1080 Ti
Limbs: 1 | Message: 42 | Match: True
Encryption : 0.011848 seconds
Decryption : 0.027494 seconds
 GPU usedd 0: NVIDIA GeForce GTX 1080 Ti
Limbs: 2 | Message: 124 | Match: True
Encryption : 0.031335 seconds
Decryption : 0.075907 seconds
 GPU usedd 0: NVIDIA GeForce GTX 1080 Ti
Limbs: 3 | Message: 1234 | Match: True
Encryption : 0.066196 seconds
Decryption : 0.157805 seconds
```

[ ]: