HIGH PERFORMANCE COMPUTING for SCIENCE & ENGINEERING (HPCSE) I

HS 2021

EXERCISE 01: AMDAHL'S LAW, ROOFLINE MODEL, CACHE

Pascal Auf der Maur

Computational Science and Engineering Lab
ETH Zürich

22.10.2021

Outline

1. Discussion Exercise 01
2. Tutorial Libraries

Amdahl’s Law

a)

Suppose you have a program where 99.99% of the runtime is parallelizable. You want to run this program on a super computer that has up to 2’000’000 cores. What is the maximum speed up you can achieve if you had no limitations of processors n? What speed up can you achieve with 20, 200, 2000, 20’000, 200’000 and 2’000’000 cores?

$$S(n) = \frac{1}{(1-p)+\frac{p}{n}} \qquad p = 0.0001$$

Cores	Speedup
20	19.96
200	196.1
2’000	1’666.81
20’000	6’666.89
200’000	9’523.85
2’000’000	9’950.25
∞	10’000

Amdahl's Law

Points: 6

3 for finding the maximum achievable speedup,

3 for getting the speedup for the corresponding number of cores n (at least 4 correct)

Amdahl's Law

b)

You are executing a code with serial fraction 0.01. However, there are parts of a program that do not benefit from increasing the number of cores n used. Assume that the parallel execution of your code requires a communication operation, which costs a constant amount of time for every core, therefore the corresponding overall time for this operation scales proportionally with the number of cores as 0.01n. What is the maximum speed up you can achieve with this constraint (and for how many cores)? If you reduce the time spent on this communication operation to 0.001n, what is the new maximum speedup?

$$T(n) = T_0 \left(\underbrace{0.01}_{\text{serial fraction}} + \underbrace{0.01n}_{\text{communication}} + \underbrace{\frac{0.99}{n}}_{\text{parallel fraction}} \right)$$

Amdahl's Law

b)

$$n_{opt} = \arg\min T(n)$$

$$\frac{\partial}{\partial n}\left(0.01 + 0.01n + \frac{0.99}{n}\right) = 0.01 - \frac{0.99}{n^2}$$

$$0.01 - \frac{0.99}{n^2} = 0 \implies n \approx 9.95$$

$$S(9) = 4.76 \quad S(10) = 4.78$$

$$n_{opt} = 10$$

Equivalent for $p = 0.001$

$$S(31) = 13.7107, S(32) = 13.7104$$

$$n_{opt} = 31$$

Amdahl's Law

b)

Points: 12

- 3 for the correct derivation of the speedup
- 6 for correctly taking the derivative of the denominator
- 3 for the correct numerical results (0.5 per result)

Amdahl's Law

c)

Upon inspection of a code, you see that 1000 of the code's operations are parallelizable and 10 are not. Suppose the time to compute one operation (both serial and parallel) is t_1.

- Compute the time $T(n)$ to execute the code with n cores.
- Using this time T (n), compute the speed-up S(n) for n = 10. Verify your answer using Amdahl's law.
- The speed up you get is true only in the case of perfect load balancing. For n = 10, re-compute the speed-up assuming one core is assigned 1.5 and 3 times more of the parallel operations.

Amdahl's Law

c)

$$T(n) = \frac{1000t}{n} + 10t$$

$$S(n) = \frac{T(1)}{T(n)} = \frac{1010t}{\frac{1000t}{n} + 10t}$$

$$S(10) = 9.1818$$

$$p = \frac{\text{parallel op}}{\text{total op}} = \frac{1000}{1010} \approx 0.990099$$

$$S(10) = \frac{1}{(1 - 0.990099) + \frac{0.990099}{10}} = 9.1818$$

Amdahl's Law

c) Calculating speedup with imbalance:

$$S(n) = \frac{T(1)}{\max(T_{core}) + T_{serial}}$$

1.5x the work:

$$S(n) = \frac{1010t}{1.5 \cdot 100t + 10t} \approx 6.3125$$

3x the work:

$$S(n) = \frac{1010t}{3 \cdot 100t + 10t} \approx 3.258$$

Amdahl's Law

c)

Points: 12

- 3 for finding the time T(n) and using it to find the speed up S(n)
- 3 for verifying the speedup using Amdahl's law
- 3 for finding the time of the slowest core for both imbalance factors
- 3 for finding the speedup for both imbalance factors

Parallel Scaling

A program simulates N particles. All particles interact with each other and, thus, the number of interactions is proportional to N^2. The runtimes of the program in seconds on P processor cores are in the table:

P\N	500	1000	1500	2000
1	6.00	30.00	72.00	120.00
4	1.50	7.50	18.00	30.00
9	0.75	3.50	9.00	20.00
16	0.50	2.15	6.00	12.00
24	0.40	1.50	4.50	10.00

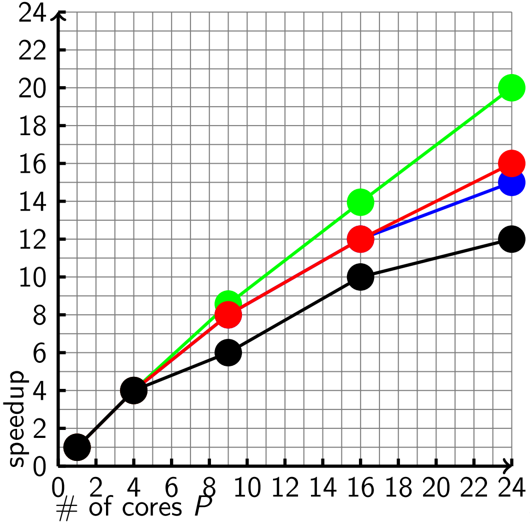
Parallel Scaling

Strong Scaling $S(p) = T(1)/T(p)$

N\P	1	4	9	16	24
$S(p) = T(1)/T(p)$	$\frac{6.0}{6.0} = 1.0$	$\frac{6.0}{1.5} = 4.0$	$\frac{6.0}{0.75} = 8.0$	$\frac{6.0}{0.5} = 12.0$	$\frac{6.0}{0.4} = 15.0$
$S(p) = T(1)/T(p)$	$\frac{30.0}{30.0} = 1.0$	$\frac{30.0}{7.5} = 4.0$	$\frac{30.0}{3.5} \approx 8.57$	$\frac{30.0}{2.15} \approx 13.95$	$\frac{30.0}{1.5} = 20$
$S(p) = T(1)/T(p)$	$\frac{72.0}{72.0} = 1.0$	$\frac{72.0}{18.0} = 4.0$	$\frac{72.0}{9.0} = 8.0$	$\frac{72.0}{6.0} = 12.0$	$\frac{72.0}{4.5} = 16.0$
$S(p) = T(1)/T(p)$	$\frac{120.0}{120.0} = 1.0$	$\frac{120.0}{30.0} = 4.0$	$\frac{120.0}{20.0} = 6.0$	$\frac{120.0}{12.0} = 10.0$	$\frac{120.0}{10.0} = 12.0$

Parallel Scaling

Strong Scaling



Parallel Scaling

Weak scaling

When using more cores we also scale the problem to only measure overhead:

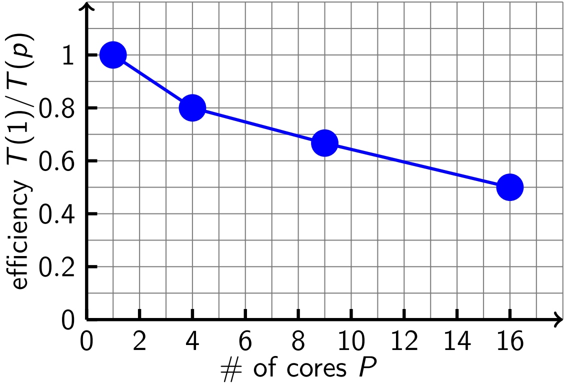
Eg. Computational complexity $O(N^2) \implies$ 4x cores and 2x N

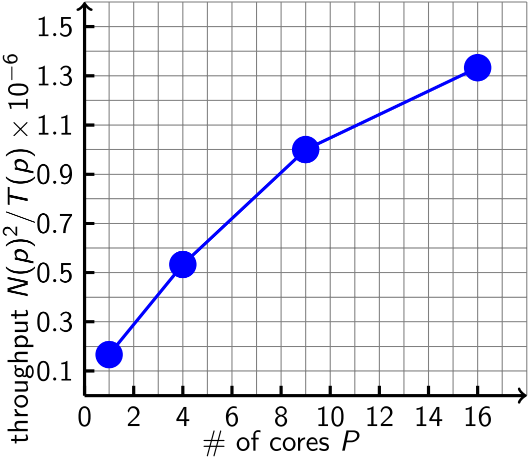
efficiency $T(1)/T(p)$ or throughput $N(p)^2/T(p)$ as metrics

P	1	4	9	16
$T(1)/T(p)$	$\frac{6.0}{6.0} = 1.0$	$\frac{6.0}{7.5} = 0.8$	$\frac{6.0}{9.0} = 0.667$	$\frac{6.0}{12.0} = 0.5$
$N(p)^2/T(p)/10^6$	$\frac{1.0}{6.0} = 0.1667$	$\frac{4.0}{7.5} = 0.533$	$\frac{9.0}{9.0} = 1.000$	$\frac{16.0}{12.0} = 1.333$

Parallel Scaling

Weak scaling





Roofline Model

Our kernel:

```c
float A[N], B[N], C[N];
...
const int P = 2;
for (int i = 0; i < N; ++i) {
    int j = 0;
    while (j < P) {
        A[i] = B[i] * A[i] + 0.5;
        ++j;
    }
    C[i] = 0.9 * A[i] + C[i];
}
```

Roofline Model

a)
What is the floating point operational intensity of the code? State all the assumptions you made and show your calculations.

Operational intensity:
Flops $= (2P + 2)N$ [Flop]

Memory operations

Without cache:

$$D = (3 \cdot P \cdot 4 + 3 \cdot 4)N \,[\text{Bytes}]$$

$$OI = (2P + 2)/(12P + 12) = 1/6 \,[\text{Flop/B}]$$

∞ cache:

$$D = (5 \cdot 4)N \,[\text{Bytes}]$$

$$OI = (2P + 2)/20 = 3/10 \,[\text{Flop/B}]$$

Roofline Model

a)

Points: 10

- 5 points 1 per correct read/write operation found
- 3 points for correct flops
- 2 points for correct operational intensity

Roofline Model

b)
For a peak performance of 409.7 GFLOP/s (single precision) and a memory bandwidth of 34 GB/s, find all positive P for which the code is memory bound. Assume an infinite cache and state further assumptions you made. Show your calculations.

$$\begin{aligned} OI_{Ridge} &= 409.7/34 = 12.05\,[\text{Flop/B}] \\ OI &\geq OI_{Ridge} \\ (2P+2)/20 &\geq 12.05 \\ P &\geq 119.5 \end{aligned}$$

Roofline Model

b)

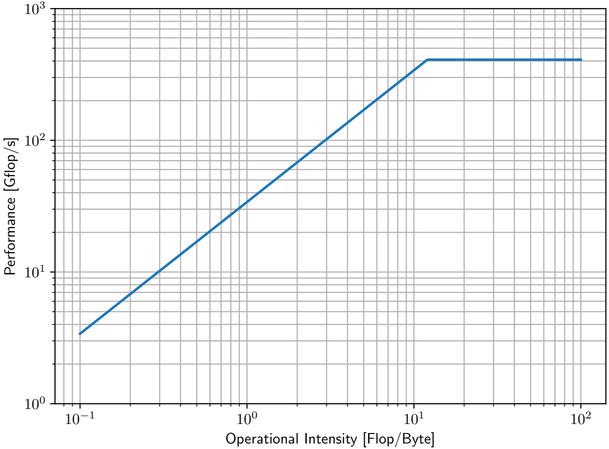
Points: 12

- 4 points for correct ridge point
- 4 points for assuming perfect caching and that A,B,C are read only once
- 4 points for correct inequality for P

Roofline Model

c)

Draw below the roofline corresponding to (b) and label the axes.



Roofline Model

c)

Points: 8

- 4 points for correct axis labels
- 2 points for correct ridge point
- 2 points for correct slope

Linear Algebra Operations

a)

We want to implement the multiplication of a square matrix $A \in \mathbb{R}^{N \times N}$ with a vector $x \in \mathbb{R}^{N}$. Notice that we have two options, to store the matrix A row-wise (row major order) or column-wise (column major order). You are provided with a skeleton code and you are asked to implement the product Ax storing A with row major order first and then with column major order. You can initialize $A_{i,j} = i + j$ and $x_i = i$. Which implementation is faster and why? What do you observe as the dimension of the matrix increases? How do you explain it?

Linear Algebra Operations

a)

Row-major

```
void Ax_row(const std::vector<double>& A,
    const std::vector<double>& x, std::vector<double>& y)
{
const size_t N = x.size();
y.assign(N, 0.0);

for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < N; j++)
        y[i] += A[i * N + j] * x[j];
}
```

Linear Algebra Operations

Assembly code Row-major (Only works on scalar values)

```
.L63:                                    // Outer loop begin
    movsd    (%rdx), %xmm1               // Load y value
    xorl     %eax, %eax
    .p2align 4,,10
    .p2align 3
.L64:                                    // Inner loop begin
    movsd    (%rcx,%rax,8), %xmm0        // load A
    mulsd    (%rsi,%rax,8), %xmm0        // multiplication with x
    addq     $1, %rax                    // increment inner index by one
    addsd    %xmm0, %xmm1                // addition
    movsd    %xmm1, (%rdx)
    cmpq     %rbx, %rax
    jne      .L64
    addq     $8, %rdx
    addq     %rbp, %rcx
    cmpq     %rdx, %rdi
    jne      .L63
```

Linear Algebra Operations

a)

Column-major

```
void Ax_col(const std::vector<double>& A,
const std::vector<double>& x, std::vector<double>& y)
{
const size_t N = x.size();
y.assign(N, 0.0);

for (size_t j = 0; j < N; j++)
    for (size_t i = 0; i < N; i++) // Changed order of for loops
        y[i] += A[j * N + i] * x[j]; // Easier to vectorize
                                     // Check assembler code with -S
                                     // Uses more SIMD registers
}
```

Linear Algebra Operations

Assembly code Column-major (Vectorized by compiler)

```
    movsd   (%r14), %xmm1            // load x value
    xorl    %eax, %eax
    unpcklpd        %xmm1, %xmm1     // unpack x value into whole reg.
.L76:                                // begin inner loop
    movupd  (%rcx,%rax), %xmm0       // load A values
    movupd  (%rdx,%rax), %xmm2       // load x values
    mulpd   %xmm1, %xmm0             // multiplication
    addpd   %xmm2, %xmm0             // addition
    movups  %xmm0, (%rdx,%rax)
    addq    $16, %rax                // increment inner index by two
    cmpq    %r9, %rax                // matches our observation
    jne     .L76
```

Linear Algebra Operations

a)

Points: 4

2 for implementing the matrix-vector multiplication.

1 for observing that the speedup is increasing as the N increases.

1 for explaining using cache line.

Linear Algebra Operations

b)

We want to compute the transpose of a square matrix $A \in \mathbb{R}^{N \times N}$ The matrix A and its inverse $A^\top$ are stored in row major order in our implementation. One way to do the transposition is the straightforward way, where we loop over all the indices and we set the corresponding (i, j) element of $A^\top$ equal to the (j, i) element of A. As you have seen in the lecture, this is not the most efficient way in terms of cache usage due to compulsory misses. A solution to this problem is to work in blocks. First implement the straightforward transposition algorithm in the provided skeleton code. Then, imitate the multiplication algorithm that was explained in classroom and use the provided skeleton code to implement a block version of the transposition algorithm. Run your code over increasing matrix dimensions and vary the block size. What do you observe? How do you explain it? In the end, the results are saved in a file. Plot the results with your favourite tool. Analyze the results and draw your conclusion.

Linear Algebra Operations

b)

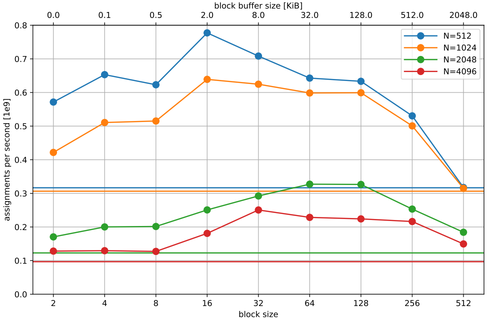
```
void transpose_block(std::vector<double>& A,
        std::vector<double>& AT, size_t blockSize)
{
    size_t N = sqrt(A.size());

    // TODO: Question 2b: Block matrix transposition
    for (size_t j = 0; j < N; j += blockSize) // Outer loops
        for (size_t i = 0; i < N; i += blockSize)
#ifdef WRITESTRIDE  // Stride in write
            for (size_t k = i; k < i + blockSize; k++)
                for (size_t l = j; l < j + blockSize; l++)
#else               // Stride in read
            for (size_t l = j; l < j + blockSize; l++)
                for (size_t k = i; k < i + blockSize; k++)
#endif
                    AT[k + l * N] = A[l + k * N]; // indeces
}
```

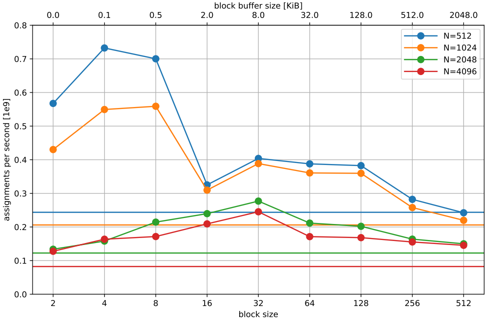
Linear Algebra Operations

b)

Continuous writes



Strided writes



Linear Algebra Operations

b)

Points: 6

4 for implementing the transposition algorithms.

1 for finding an optimal block size.

1 for plotting the results and making at least one relevant observations.

Linear Algebra Operations

c)

We want to compute the product of two square matrices $A, B \in \mathbb{R}^{N \times N}$ The matrices A, B are stored in row major order in our implementation. One way to do the multiplication is the straightforward way,

$$C_{i,j} = \sum_k A_{i,k} B_{k,j} \; .$$

As discussed in the previous subquestion, you have already seen the reasons this is not the most efficient way of implementing the product. First implement the straightforward multiplication algorithm in the provided skeleton code. Then implement the block-multiplication algorithm that was explained in the classroom. Finally, implement a second version of the block-multiplication algorithm were the matrix B is stored in column major order. You can initialize $B_{i,j} = 2i + j$. Run you code over increasing matrix dimensions and vary the block size. What do you observe? How do you explain it? In the end, the results are saved in a file. Plot the results with your favourite tool. Analyze the results and draw your conclusion. Which one is the most efficient algorithm? Explain in terms of cache usage.

Linear Algebra Operations

c)

Naive implementation

```
for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < N; j++)
        for (size_t k = 0; k < N; k++)
            C[i * N + j] += A[i * N + k] * B[k * N + j];
```

Linear Algebra Operations

c)

Block matrix-matrix multiplication - B in row major

```
for (size_t K = 0; K < N; K += blockSize)
    for (size_t I = 0; I < N; I += blockSize)
        for (size_t J = 0; J < N; J += blockSize)

            for (size_t i = I; i < I + blockSize; i++)
                for (size_t j = J; j < J + blockSize; j++)
                    for (size_t k = K; k < K + blockSize; k++)

                        C[i * N + j] += A[i * N + k] * B[k * N + j];
```

Linear Algebra Operations

c)

Block matrix-matrix multiplication - B in column major

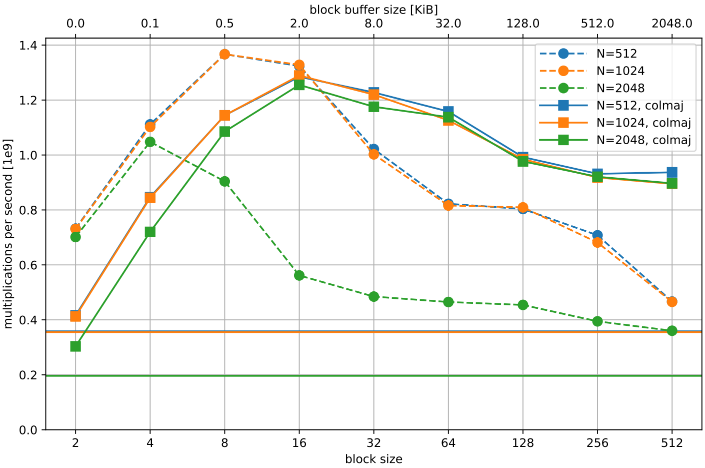
```
for (size_t K = 0; K < N; K += blockSize)
    for (size_t I = 0; I < N; I += blockSize)
        for (size_t J = 0; J < N; J += blockSize)

            for (size_t i = I; i < I + blockSize; i++)
                for (size_t j = J; j < J + blockSize; j++)
                    for (size_t k = K; k < K + blockSize; k++)

                        C[i * N + j] += A[i * N + k] * B[k + N * j];
```

Linear Algebra Operations

c)



Linear Algebra Operations

c)

Points: 10

4 for implementing the matrix-matrix multiplication algorithm, naive + B row-major.

2 for implementing the matrix-matrix multiplication algorithm, B column- major.

2 for plotting the results.

1 for showing that B column major is more efficient.

1 for explaining that B column major is more efficient.

Cache size and cache speed

a)

```
grep . /sys/devices/system/cpu/cpu0/cache/index*/*
```

Model	L1D	L1I	L2	L3	Cache line
XeonGold_5188 (Euler V)	32 kB	32 kB	1024 kB	16896 kB	64 B
XeonGold_6150 (Euler IV)	32 kB	32 kB	1024 kB	25344 kB	64 B
XeonE3_1585Lv5 (Euler III)	32 kB	32 kB	256 kB	8192 kB	64 B

Cache size and cache speed

Initialization

```
double measure(int N, int mode) {
    if (mode == 0) {
        // Random
        sattolo(a, N);
    } else if (mode == 1) {
        // Increment by one
        for (int i = 0; i < N; ++i)
            a[i] = (i + 1) % N;
    } else if (mode == 2) {
        // One cache-line at a time
        for (int i = 0; i < N; ++i)
            a[i] = (i + 64 / sizeof(int)) % N;

  ...
}
```

Cache size and cache speed

Sattolo

```cpp
void sattolo(int *p, int N) {
    // Initialization
    for (int i = 0; i < N; ++i)
        p[i] = i;
    // Draw a new entry from remaining ones
    for (int i = 0; i < N - 1; ++i)
        std::swap(p[i], p[i + 1 + rand() % (N - i - 1)]);
}
```

Cache size and cache speed

Measurement

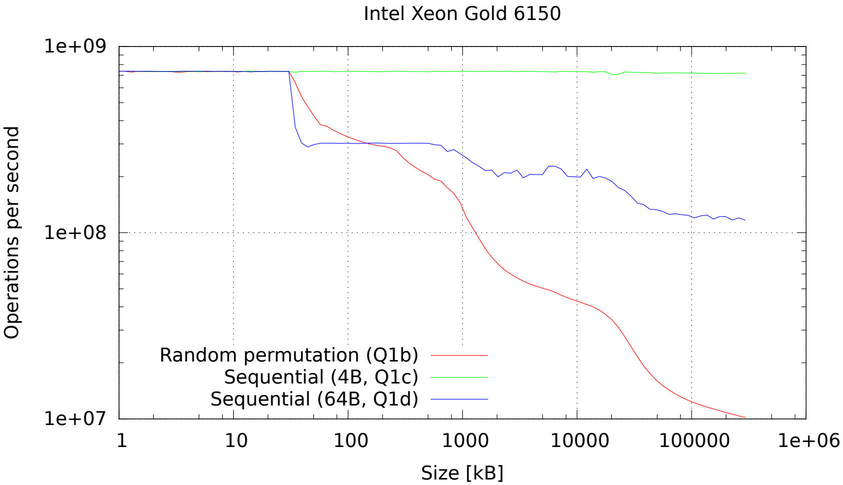
```cpp
    ...
    auto start = std::chrono::steady_clock::now();
    int k = 0;  // Start node.
    for (int i = 0; i < M; ++i)
        k = a[k];
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> diff = end - start;

    // Store into a volatile variable to prevent optimizing away the loop above.
    volatile int tmp = k;
    // "Use" the value to suppress -Wunused-variable.
    (void)tmp;

    // TODO: Question 3b: Return execution time in seconds.
    return diff.count();
}
```

Linear Algebra Operations

c)


Intel Xeon Gold 6150
Operations per second
1e+09
1e+08
1e+07
1e+06
Size [kB]
Random permutation (Q1b)
Sequential (4B, Q1c)
Sequential (64B, Q1d)