REGIONALES HOCHSCHUL-
RECHENZENTRUM KAISERSLAUTERN

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

RHRK-Tutorial

Vectorization: An elaborated example

Course instructor: Gabriele Monaco
In charge: Dr. Josef Schüle, RHRK

Measuring performances

- **GFLOPS**
  - **Flops is Floating Point Operations per Second, 1 GFlop is equivalent to 1.000.000.000 operations per second**
  - **Useful to compare performances of different systems while doing multiplications and sums/subtractions**
- **Peak performance**
  - **The best operation is FMA (fused multiply add) computing multiplication and addition in the same cycle (2 operations)**
  - **Without vectorization: 1 FMA unit per core 2 GHz (1/cycle time), AVX provides additional FMA units, Skylake can have 2 AVX512 units per core**
    - 4 Gflops (without vectorization)
    - 16 Gflops (double precision on AVX/AVX2)
    - 32 Gflops (double precision on AVX512)
    - 64 Gflops (double precision on Skylake)

Measuring performances

- **We can set up a benchmark to measure Gflops**
  - Run our function for REP times
  - An extra run ensures the cpu is up and running fine
  - Measure time before and after (exclude initialization)
  - The result will be:
    - `#operations * #repetitions / Elapsed time (ns)`
  - Should adjust the size of the arrays and number of repetitions to ensure the time is measured with right precision

```
void func(float *a, float *b, float *c) {
#pragma ivdep
    for(int i=0; i<SIZE; i++)
          c[i] *= a[i] + b[i];
}


int main(int argc, char *argv[]) {
     long start, end, elapsed;
     float gflops;
     float a[SIZE], b[SIZE], c[SIZE];
     func(a, b ,c); //dumb run

     start = gettime_ns();
     for(int i=0; i<REP; i++)
          func(a, b ,c); //any workload
     end = gettime_ns();

     elapsed = end - start;
     gflops = (2.0*SIZE) * REP / elapsed;
     printf("%f GFlops\n", gflops);
     return (int)c[SIZE/2];
}
```

Matrix-vector multiplication: setup

- Data type parametrized in a macro (to be changed easily)
  - Here just using double
- The size of the problem is the one of the matrix
  - ROWS*COLS
  - Keep it small to exploit cache
- Define and initialize matrices and vectors
  - The matrix fits in a linear array ROWS*COLUMNS
  - COLBUF is 0 (remember about this for later!)

```
#define ROW 64
#define COL 63
#define COLBUF 0
#define COLWIDTH (COL + COLBUF)
#define REPEATNTIMES 10000000
#define FTYPE double
unsigned int inc_i = 1;
unsigned int inc_j = 1;

int main(int argc, char **argv) {
    int i;
    double duration = 0.0;
    FTYPE a[ROW*COLWIDTH];
    FTYPE b[ROW] = { 0.0 };
    FTYPE x[COLWIDTH];
    init_matrix(ROW, COL, 1.0, a);
    init_vector(COL, 3.0, x);
    // Do the measurement
    start_timer();
    for (i = 0; i < REPEATNTIMES; i++) {
        matvec(ROW, COLWIDTH, a, b, x);
    }
    duration = stop_timer();
    printf("Elapsed time = %lf s\n", duration);
    printf("GigaFlops = %f\n",
           (((double)REPEATNTIMES *
           (double)ROW * (double)COLWIDTH * 2.0) /
               duration) / 1.0e9);
    return 0;
}
```

Matrix-vector multiplication

- Increments come from another file
  - Set to 1 there, but the compiler may not know and produce general code!
- Simple matrix vector multiplication
- No care has been taken, the compiler will most likely not vectorize

```
#define ROW 64
#define COL 63
#define COLBUF 0
#define COLWIDTH (COL + COLBUF)
#define REPEATNTIMES 10000000
#define FTYPE double

extern unsigned int inc_i;
extern unsigned int inc_j;

void matvec(int rows, int cols,
            FTYPE *a, FTYPE *b, FTYPE *x) {
    long i, j;
    for (i = 0; i < rows; i += inc_i) {
        for (j = 0; j < cols; j += inc_j) {
            b[i] += a[i*cols+j] * x[j];
        }
    }
}
```

Matrix-vector multiplication: unit-stride

- Include the increment here
  - The compiler knows now the loop's stride is always going to be 1 and can build better code
- Improved performances, although probably still not vectorizing

```
#define ROW 64
#define COL 63
#define COLBUF 0
#define COLWIDTH (COL + COLBUF)
#define REPEATNTIMES 10000000
#define FTYPE double

void matvec(int rows, int cols,
            FTYPE *a, FTYPE *b, FTYPE *x) {
    long i, j;
    int inc_i = 1, inc_j = 1;
    for (i = 0; i < rows; i += inc_i) {
        for (j = 0; j < cols; j += inc_j) {
            b[i] += a[i*cols+j] * x[j];
        }
    }
}
```

Matrix-vector multiplication: ivdep

- Tell the compiler we are sure about our pointers
  - They do not overlap (used both pragma and restrict keyword here), the compiler can simplify the code a lot with just this assumption
- Finally we should see it vectorized
  - To really have good performance, remember to tell the compiler to use AVX!

```
#define ROW 64
#define COL 63
#define COLBUF 0
#define COLWIDTH (COL + COLBUF)
#define REPEATNTIMES 10000000
#define FTYPE double

void matvec(int rows, int cols,
            FTYPE * restrict a,
            FTYPE * restrict b,
            FTYPE * restrict x) {
    long i, j;
    int inc_i = 1, inc_j = 1;
#pragma ivdep
    for (i = 0; i < rows; i += inc_i) {
        for (j = 0; j < cols; j += inc_j) {
            b[i] += a[i*cols+j] * x[j];
        }
    }
}
```

That's the biggest improvement, but something can still be done..

Matrix-vector multiplication: padding

- **The size of the column is a little odd**
  - Every new column has 63 as offset from the previous, memory accesses will soon be disaligned
  - By padding each column (using here COLBUF as 1), we can have the same alignment for all rows, without changing the size of our matrix
  - Of course we are going to use a little more memory but access will be faster (we would be loading that memory anyways)

```
#define ROW 64
#define COL 63
#define COLBUF 1
#define COLWIDTH (COL + COLBUF)
#define REPEATNTIMES 10000000
#define FTYPE double

void matvec(int rows, int cols,
            FTYPE * restrict a,
            FTYPE * restrict b,
            FTYPE * restrict x) {
  long i, j;
  int inc_i = 1, inc_j = 1;
#pragma ivdep
  for (i = 0; i < rows; i += inc_i) {
    for (j = 0; j < cols; j += inc_j) {
      b[i] += a[i*(cols+COLBUF)+j] * x[j];
    }
  }
}
```

Matrix-vector multiplication: alignment

- The data is still not aligned
  - We need to explicitly create aligned pointers and inform the compiler about it
  - We are going to use 64 as that's the size of AVX512 registers, with AVX or AVX2 an alignment of 32 would still do the job
  - On icc the use of the appropriate pragma may improve results
- To have the absolute best results, also optimize function calls (-flto/-ipo)

```
/* While defining them */
FTYPE a[ROW*COLWIDTH]
__attribute__((aligned(64)));
FTYPE b[ROW] __attribute__((aligned(64)));
FTYPE x[COLWIDTH] __attribute__((aligned(64)));
```

```
  /* While using them */
  __builtin_assume_aligned(a, 64);
  __builtin_assume_aligned(b, 64);
  __builtin_assume_aligned(x, 64);

#pragma ivdep
#pragma vector aligned
  for (i = 0; i < rows; i += inc_i) {
     for (j = 0; j < cols; j += inc_j) {
       b[i] += a[i*(cols+COLBUF)+j] * x[j];
     }
  }
```

Matrix-vector multiplication: results

	Intel	gcc	Intel AVX512	gcc AVX512
basic	2.360123	2.524665	2.451304	2.821595
unit-stride	7.589429	2.710265	15.853811	2.83772
ivdep	7.393366	2.697124	21.877784	2.844206
restrict	7.596136	5.444614	15.556108	14.047247
padding	9.262423	5.522359	27.706556	22.037044
align	9.218424	5.541915	27.241071	21.616835
align_fopt	10.767121	5.480334	125.550949	39.760351

- High Performance Computing on Elwetritsch

Vielen Dank
Thank You