Some Performance Optimizations for Embedded Systems

Dr. Jason Forsyth

Department of Engineering
James Madison University

Overview

- Moving from desktop/prototype to embedded implementation can be challenging; limited energy, data, and CPU cycles
- Calculations performed on different architecture may be significantly slower and/or infeasible on chosen platform.
- Several non-compiler arithmetic optimizations are available

Atmel 32u4

- Up to 16 MIPS; 16/32KB Program Memory; 1.25/2.5 KB RAM
- No floating point unit; no cache;
- 8 bit ALU but can calculate results of additional length via compiler unroll
- Native Data Sizes: int (2 bytes), long/float/double (4 bytes), long long (8 bytes)

Reducing Time Awake

- To achieve battery lifetimes must minimize the amount of time that the processor is awake.
- Major tasks: count steps, display steps, store data…etc.
- How long do each of these take? How can their efficiency be improved?
- Recall speeds: register < flash < SPI < i2c…etc.; int < float

Timing Your Code

- How long does a certain operation take? Use the internal cycle counter to find out.

Use micros() call

```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = micros();

  Serial.println(time);
  delay(1000);
}
```

```
long startTime = micros();
for (int i = 0; i < LOOPS; i++)
{
  //do something
}
long endTime = micros();

long delta = endTime - startTime;

Serial.print("Direct Timing: ");
Serial.println(delta);
```

Start Time Stamp

Do it “many” times to get accurate timing

End Time Stamp

Ensure that whatever code is being timed can be “optimized away” by the compiler. Must do “something” with the final result. Default compile flag is –Os (which is –O3 but for size)

Major Bottlenecks in Code

- Memory operations: read/writing to flash is slower than RAM; accessing any external I/O is significantly slower; excessive reads/writes when not necessary

- Arithmetic operations: excessive calculations that aren’t necessary (values that never change); expensive operators;

- Algorithmic complexity: bad data structure; poor solution to problem

Measuring the Impact of float/int operations

```
#define LIMIT 256
#define TYPE float

TYPE data[LIMIT];
TYPE result[LIMIT];

long startTime = micros();
TYPE sum = 0;
for (int i = 0; i < LIMIT; i++)
{
  sum = result[i] = (10 * data[i]);
}

long endTime = micros();

long delta = endTime - startTime;

Serial.print("End: ");
Serial.println(delta);
Serial.println(sum);
```

- Many digital filter (and desktop code) utilize floating point operations. However these are not naturally supported.
- Perform floating and integer calculation of arrays of various lengths and compare execution time.
- Code can do "dummy" work but must flush/use result so not removed by the compiler.
- Examine run time, program, and data size

Program and Data Size

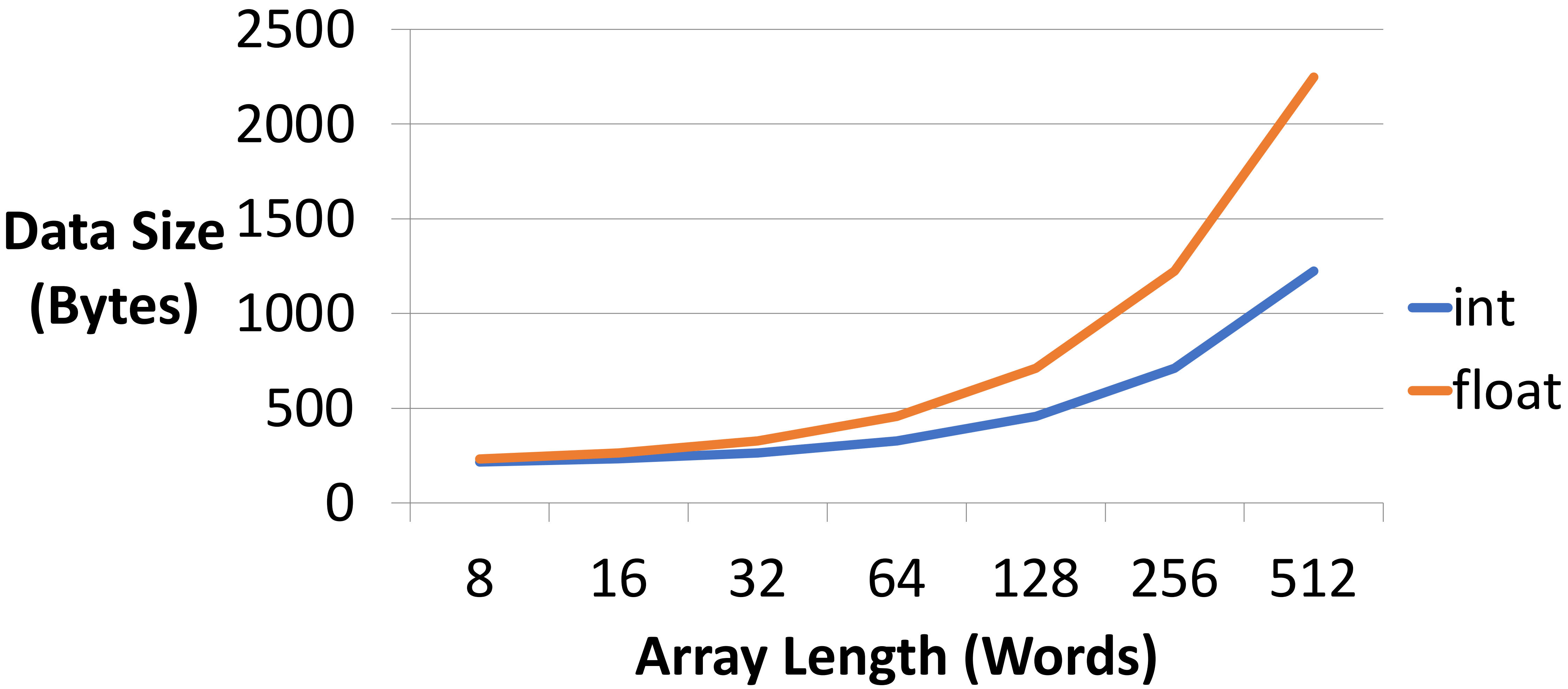
```
#define LIMIT 256
#define TYPE float

TYPE data[LIMIT];
TYPE result[LIMIT];

long startTime = micros();
TYPE sum = 0;
for (int i = 0; i < LIMIT; i++)
{
  sum = result[i] = (10 * data[i]);
}

long endTime = micros();

long delta = endTime - startTime;

Serial.print("End: ");
Serial.println(delta);
Serial.println(sum);
```

		Array	**Size**		
Program Size (bytes)	**8**	**16**	**32**	**64**	**128**
int	1948	1948	1948	1948	1948
float	1950	1950	1950	1950	1950

Data Size (bytes)	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
int	216	232	264	328	456	712	1224	2248
float	232	264	328	456	712	1224	2248	

- Program size is almost equal
- Data size grows faster (float is 2x bytes)
- Data Size = constant allocation + arrays

Program and Data Size



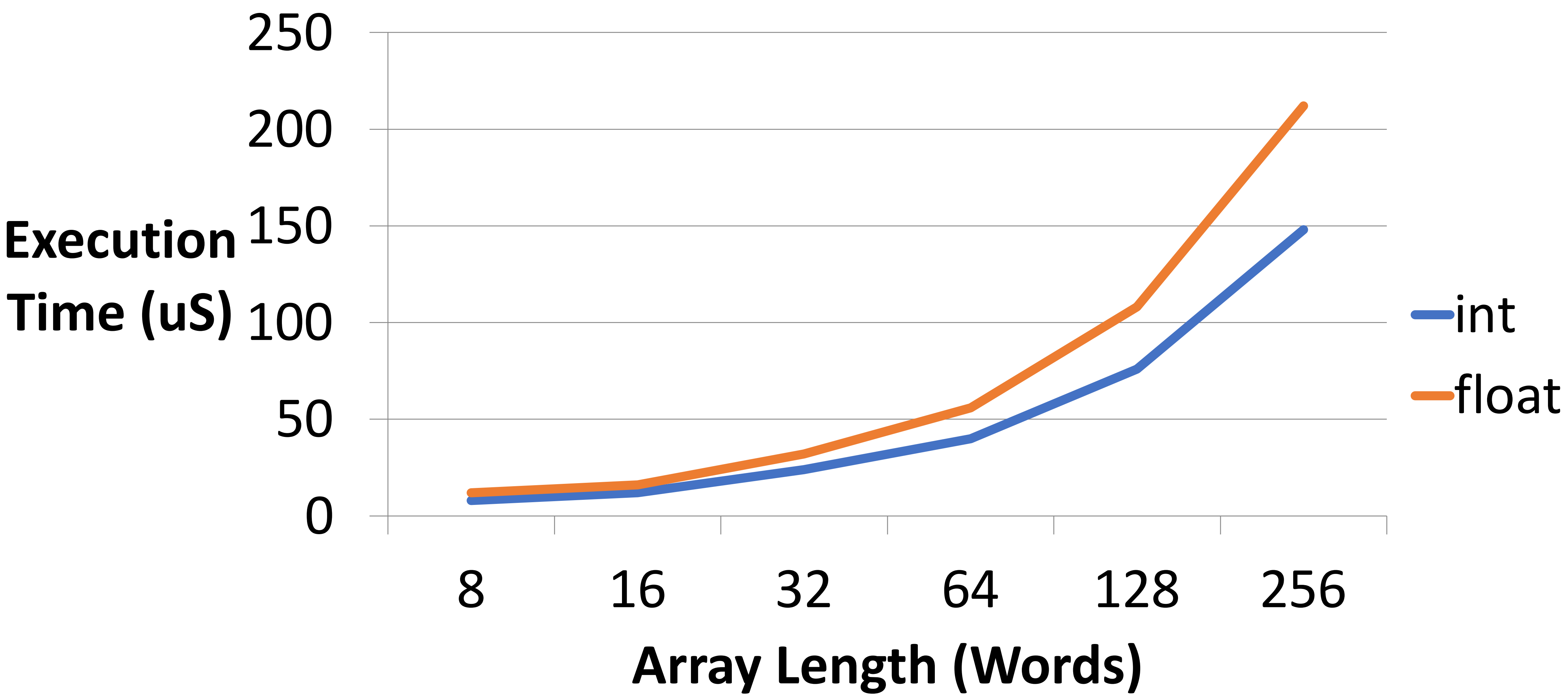
Execution Time

```
#define LIMIT 256
#define TYPE float

TYPE data[LIMIT];
TYPE result[LIMIT];
long startTime = micros();
TYPE sum = 0;
for (int i = 0; i < LIMIT; i++)
{
  sum = result[i] = (10 * data[i]);
}

long endTime = micros();

long delta = endTime - startTime;

Serial.print("End: ");
Serial.println(delta);
Serial.println(sum);
```

Execution Time (us)	8	16	32	64	128	256	512	1024
int	8	12	24	40	76	148	292	N/A
float	12	16	32	56	108	212	N/A	
Increase	50%	33%	33%	40%	42%	43%		

- Exec Time = constant stuff + loop time
- Floating point execution time is 30-50% slower
- Only doing 1 operation in this code. Would be worse if there were more…

Execution Time



So floats are bad… What to do?

- Know the cost and use them as little as possible.
- Move to a different processor? Is it worth the cost?
- “Intergerize” calculations

“Integerizing” Calculations

- Floating point calcs can be turned into integers by moving the decimal point and then dividing the result

- Float: $y = 3.1435x + 3$
- Int = $y = (31435 * x)/10000 + 3$

- Provides the “same” result but with potential loses due to integer multiplication and division

“Integerizing” Calculations

- Consider a digital filter: $y(x) = -2.386987234 * x[n] + 3.618872342 * x[n-1] + 4.125872342 * x[n-2]$
- Can be integerized with [-23870,36189,41259].
  - $y(x) = (-23870 * x[n] + 36189 * x[n-1] + 41259 * x[n-2])/10000$
- Cannot expand floats too many places as may result in overflow multiplication operation (n bits * n bits = 2*n bit result).
- Largest integer variable is *long long* at 8 bytes

“Integerizing” Calculations

- Decimal expansion is limited and will result in approximation errors
- Amount of acceptable error depends on application and filter structure (IIR may be problematic)

+/- 5E4			
X	**Float calc**	**Int calc**	**Abs Error**
15257	300010.9882	300013.1759	2.187719967
37431	240884.8791	240886.7805	1.90144279
48710	-17312.68037	-17312.512	0.168366942
37315	31688.13117	31688.4346	0.303433825
-8745	276615.662	276617.6136	1.951557713
36939	-26736.60544	-26736.5646	0.040836222

+/- 1E6			
X	**Float calc**	**Int calc**	**Abs Error**
705504	-1412453	-1412460	-7.03846
43352	4601.075	4601.341	0.265586
27798	107205.8	107206.7	0.814765
1814	289297.2	289299.3	2.082751
40475	89080.86	89081.71	0.849333
35666	163835.5	163836.8	1.260321

Accessing Memory

- Another significant delay to access memory that is ”far away” or repeated access where not necessary

- Many filters and buffers may have data that is “shifted down” for calculation

```
for (int i = 0; i < LOOPS; i++)
{
  //move the data around to do the calculation
  sum = -2 * data[0] + data[1] + 4 * data[2];

  //shift the data down
  for(int j=0;j<LEN-1;j++)
  {
    data[j] = data[j+1];
  }
}
```

Circular versus Direct Buffers

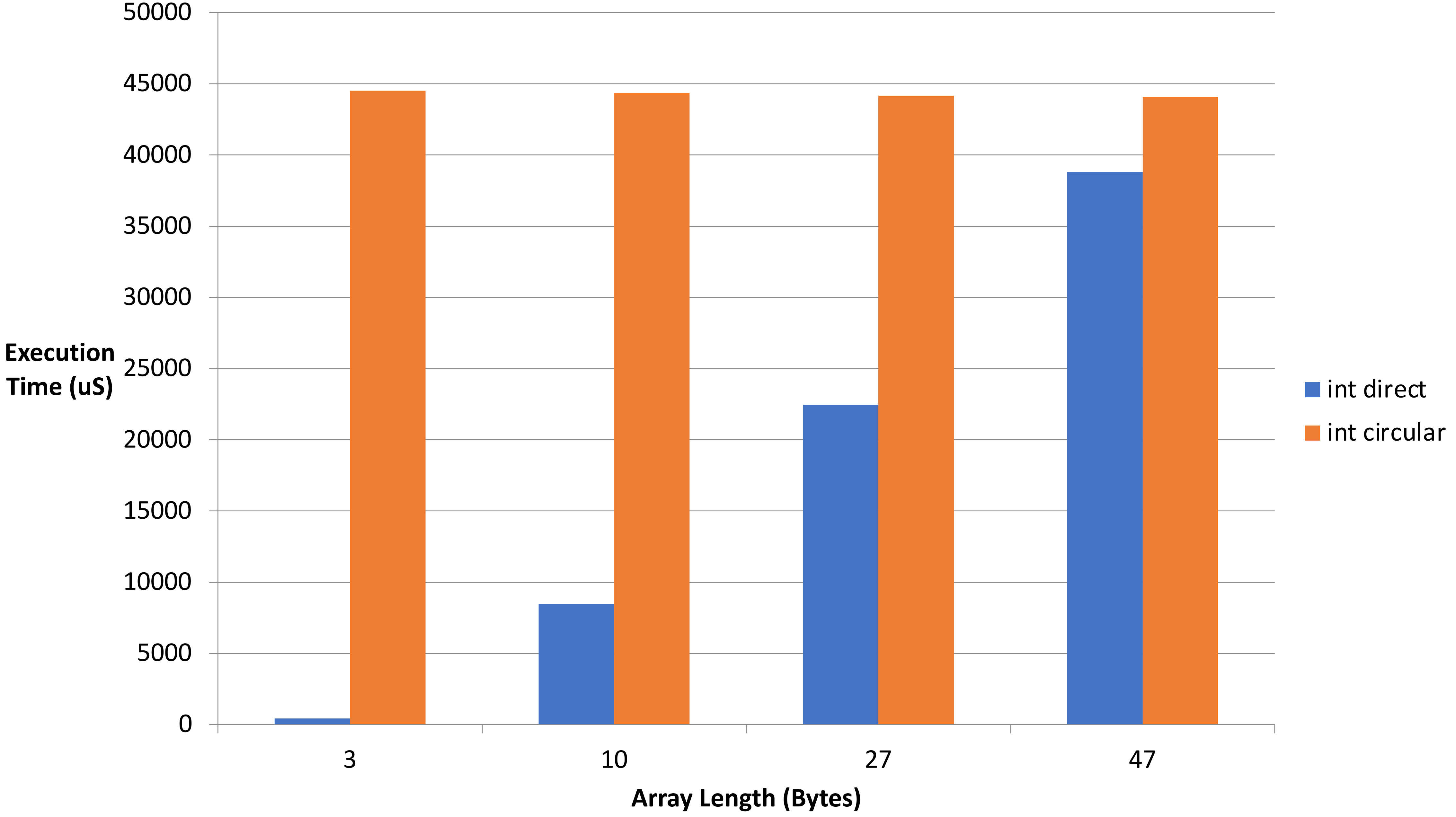
- Two approaches: move the data (direct) or move the calculation around (circular)
- Can use modulus to adjust where calculation occurs. Always make relative to some header/pointer

```
for (int i = 0; i < LOOPS; i++)
{
  //move the calculation around on the data
  sum = -2 * data[counter % LEN] + data[(counter + 1) % LEN] + 4 *
  data[counter%LEN] = 0x37; //add some new data
  counter++;
}
```

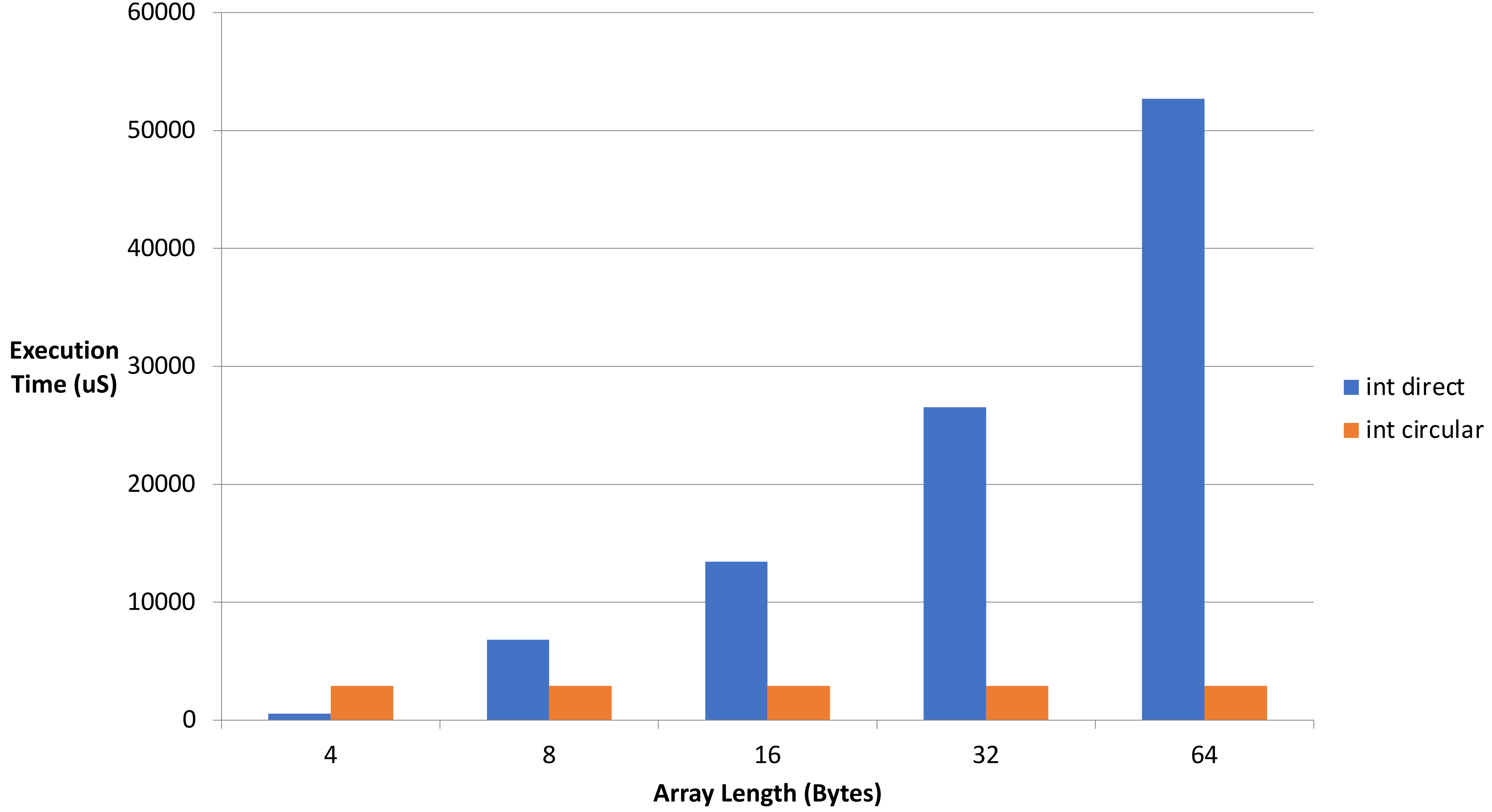
Comparing Circular and Direct Buffer Implementations

```
TYPE sum=0;
long startTime = micros();
for (int i = 0; i < LOOPS; i++)
{
  //move the data around to do the calculation
  sum = -2 * data[0] + data[1] + 4 * data[2];

  //shift the data down
  for(int j=0;j<LEN-1;j++)
  {
    data[j] = data[j+1];
  }
}
long endTime = micros();

long delta = endTime - startTime;
```

```
int counter = 0;
sum = 0;
startTime = micros();
for (int i = 0; i < LOOPS; i++)
{
  //move the calculation around on the data
  sum = -2 * data[counter % LEN] + data[(counter + 1) % LEN]
        + 4 * data[(counter + 2) % LEN];
  data[counter%LEN] = 0x37; //add some new data
  counter++;
}
endTime = micros();

delta = endTime - startTime;
```



50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
Execution Time (uS)
3
10
27
47
Array Length (Bytes)
int direct
int circular



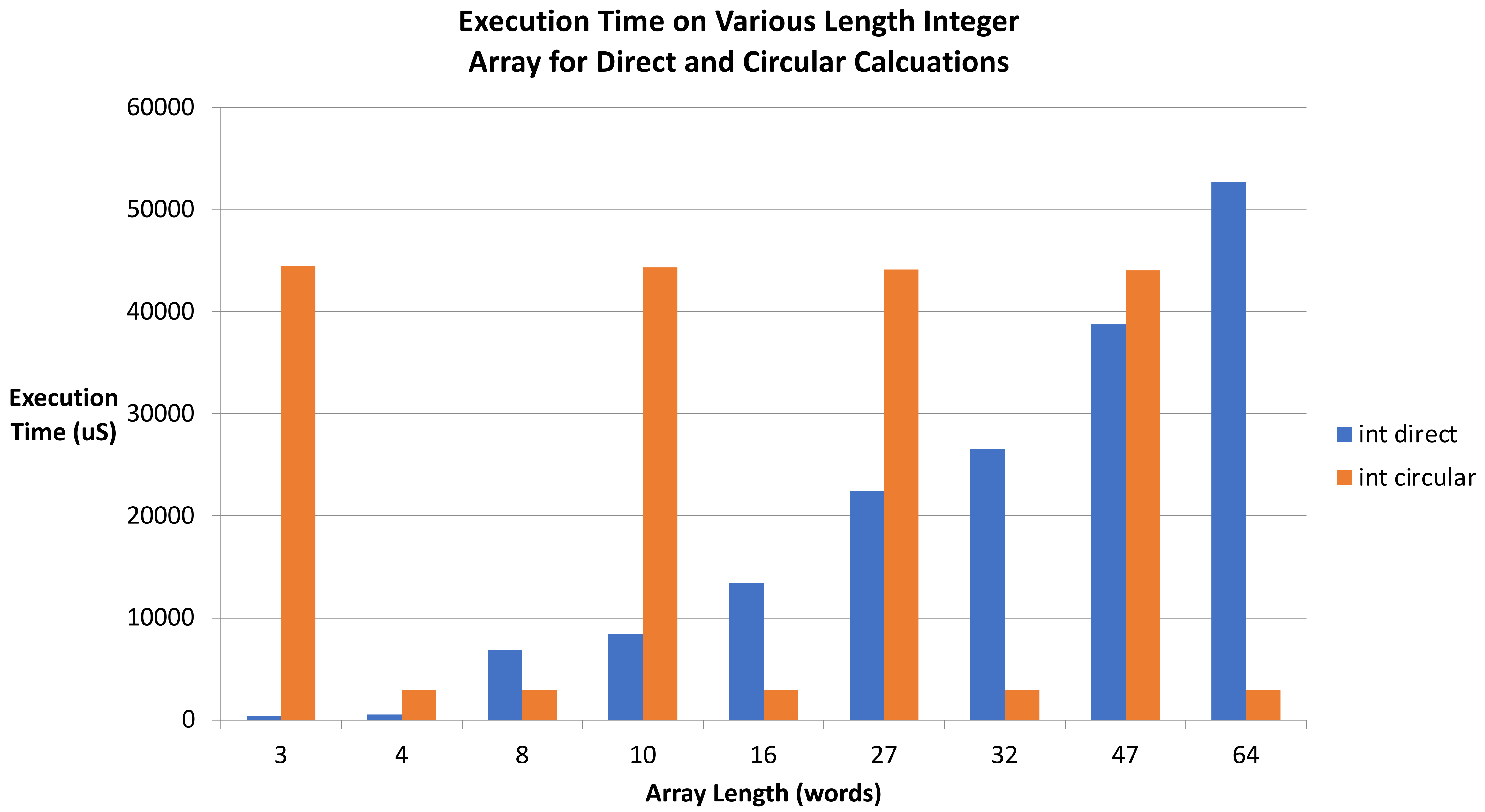
60000
50000
40000
30000
20000
10000
0
Execution Time (uS)
4
8
16
32
64
Array Length (Bytes)
int direct
int circular

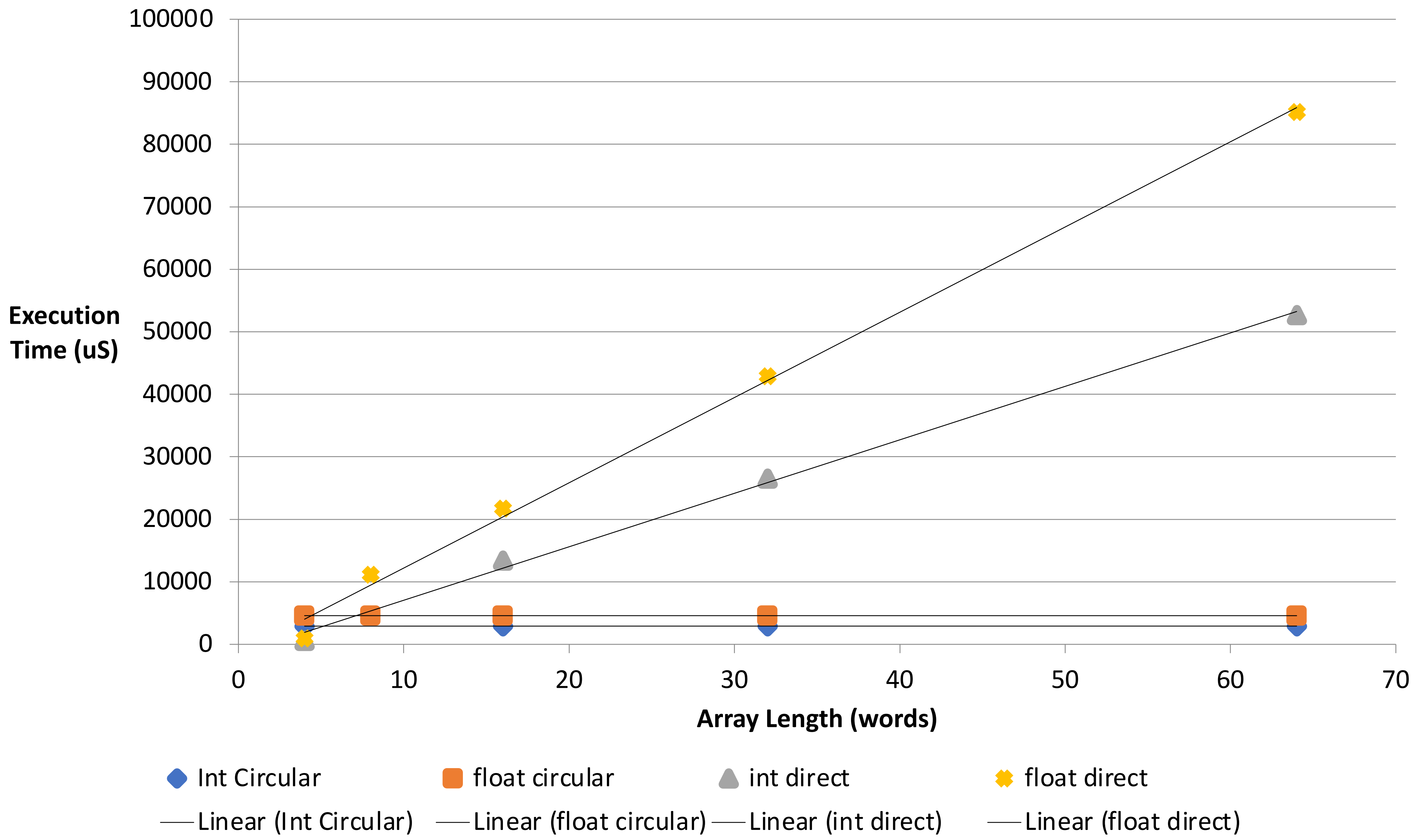
Comparing Circular and Direct Buffer Implementations

- In one slide direct is better, in the other circular is better…?

- Examine the array lengths. Cause significant difference between the algorithms.

- Modulus on 2^N value is a logical AND


Execution Time on Various Length Integer Array for Direct and Circular Calcuations
Execution Time (uS)
60000
50000
40000
30000
20000
10000
0
3
4
8
10
16
27
32
47
64
Array Length (words)
int direct
int circular


Execution Time (uS)
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
10
20
30
40
50
60
70
Array Length (words)
Int Circular
float circular
int direct
float direct
Linear (Int Circular)
Linear (float circular)
Linear (int direct)
Linear (float direct)

Comparing Circular and Direct Buffer Implementations

- Calculation of modulus on not 2^N values “swamps” any benefits of the circular buffer
- Circular buffer is constant time operation O(1)
- Direct buffer is O(N)
- Float still more expensive than int operation.

How many optimizations to implement?

- Many (infinite) things to be done to make faster/better?
- Final requirement will be on step counting and current requirements
- $i_{avg} = 0.66 * i_{active} + 0.33 * i_{idle}$
- Reduction in running current is 2x more effective than reduction in sleep.
  - $\Delta i_{avg} = 0.66 * \Delta i_{active} + 0.33 * \Delta i_{idle}$

Other energy considerations

- Leave GPIO in “good” state. i2C pins left at 0V will drain current while asleep. Ensure all transactions are complete.

- Don’t wake up too frequently. Use the watermark to delay running. Also, buffer EEPROM operations. Page write takes 4ms regardless.